Datenstrukturen

Ein Text zur Vorlesung "Effiziente Algorithmen" von Prof. E. Mayr

⋆⋆⋆

Jens O. Ernst

Technische Universität München

22.07.1995

Inhaltsverzeichnis

Kapitel 1

Primitive Datenstrukturen

1.1 Grundlegende Datentypen

Als Grunddatentypen verwenden wir:

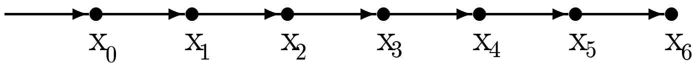
- BOOLEAN
- NAT
- INT
- FLOAT
- CHAR
- STRING
- POINTER TO <type>
- STRUCT (Geflechte)
- ARRAY (Reihungen, Felder)

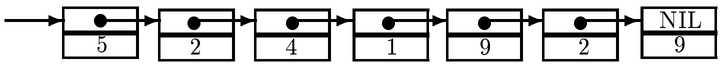
Aus diesen lassen sich folgende Datenstrukturen aufbauen:

- Listen
- Mengen
- Relationen (Menge von Tupeln)
- Funktionen (rechtseindeutige Relationen)

1.2 Listen

Definition (Liste): Eine Liste ist eine Folge beliebiger und nicht notwendig paarweise verschiedener Elemente, deren Reihenfolge berücksichtigt wird. Notation: $[x_0, x_1, \cdots, x_{n-1}]$ Die Länge der Liste q wird mit $|q|$ bezeichnet, die leere Liste mit $q' = [\ ]$. Graphische Darstellung:





Operationen auf Listen:

- *is-in-list*
- *insert*
- *delete*
- *sublist*
- *concatenate*

Eine Teilliste (sublist) ist dabei nicht notwendig zusammenhängend in der ursprünglichen Liste.

Spezielle Listen:

Folgende häufig benötigte Datenstrukturen sind Spezialfälle von Listen:

1. **queue (Warteschlange):** Es stehen nur die Operationen *insert* und *delete* zur Verfügung. *Insert* (*enqueue*) ist nur am einen Ende der Warteschlange möglich, und *delete* (*dequeue*) nur am anderen Ende. Diese Organisation heißt auch FIFO–Struktur.

2. **Stack (Keller):** Es stehen nur *insert* und *delete* zur Verfügung. Die *insert*– (bzw. *PUSH*–) Operation sowie die *delete*– (bzw. *POP*–) Operation sind nur am selben fest vorgegebenen Ende des Stack möglich. Es liegt eine LIFO–Struktur vor.

3. **Deque (double queue):** *insert* und *delete* sind jeweils und unabhängig voneinander an beiden Enden der Liste ausführbar.

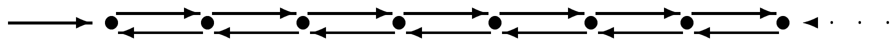
Bemerkung: Eine Warteschlange kann mit zwei Stacks simuliert werden und ein Stack durch zwei Warteschlangen. Je nach Implementierung hat dabei eine der Operationen *insert* bzw. *delete* die Komplexität $O(n)$, die andere $O(1)$.

Beweis: Simulation einer Warteschlange durch zwei Stacks:
Der erste Stack enthält die Elemente der Warteschlange, wobei einmal festgelegt wird, ob das oberste Kellerelement das erste oder das letzte Element der Warteschlange darstellt. Der andere Stack wird als Zwischenspeicher verwendet. Nur für eine Operation wird er benötigt, und zwar für *delete*, falls das hinterste Element oben auf dem Stack liegt, andernfalls für *insert*. Sei das hinterste Warteschlangenelement oben auf dem Stack. Für *insert* ist nur eine einfache *PUSH*–Operation auszuführen, für *delete* muß das unterste der n Kellerelemente entfernt werden. Daher sind die $(n-1)$ ersten Elemente vom ersten Stack zu nehmen und (in umgekehrter Reihenfolge) auf den zweiten Stack zu legen, dann ist Stack 1 vollständig zu löschen, und die Elemente auf Stack 2 sind wieder auf Stack 1 zu schreiben — unter erneuter Umkehrung der Reihenfolge. Diese Operation hat also Komplexität $O(n)$. Bei umgekehrter Speicherung ist das Vorgehen ähnlich. Ebenso geht man vor, um einen Stack mit zwei Warteschlangen zu Implementieren. *q.e.d.*

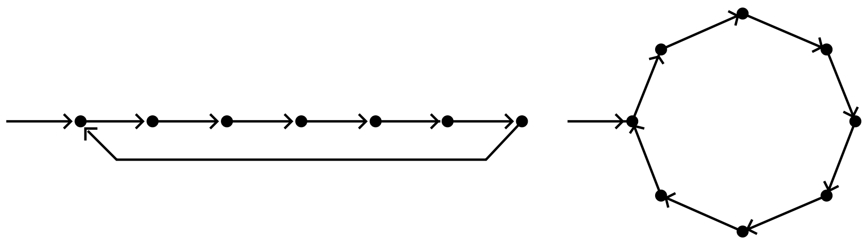
Implementierung von Listen:

1. **Einfach verkettete lineare Liste:**

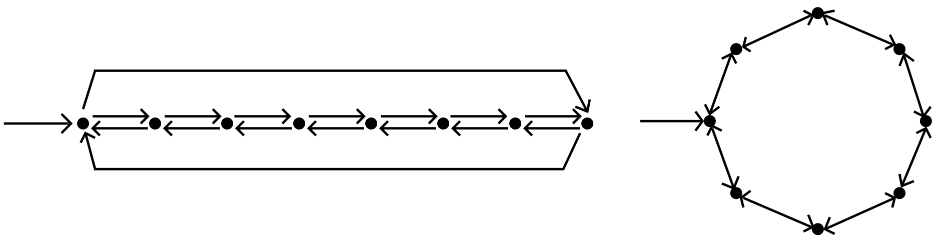
2. **Doppelt verkettete lineare Liste:**



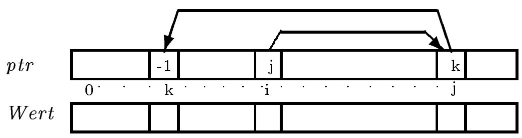
3. **Einfach verkettete zirkuläre Liste:**



4. **Doppelt verkettete zirkuläre Liste:**



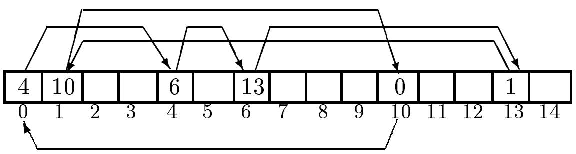
Bemerkung: Eine Liste kann mittels mehrerer (1–dimensionaler) Arrays implementiert werden. Die Reihung '*ptr*' enthält die Pointer, die Reihung '*Wert*' die Datenelemente. Dabei ist $ptr[i]$ der Index des Nachfolgers des Elements mit Datenwert $Wert[i]$.



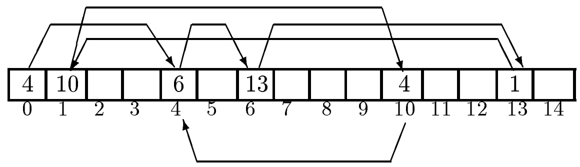
Die Elemente in Listenreihenfolge haben im Array '$Wert$' die Indizes

$$ptr[0], ptr[ptr[0]], \cdots$$

Der Indexwert (-1) stellt den NIL–Pointer dar.
Es läßt sich auch eine einfach verkettete zirkuläre Liste implementieren. Dieser Fall liegt genau dann vor, *wenn an einer erreichbaren Position in ptr der Wert* 0 *vorkommt*.

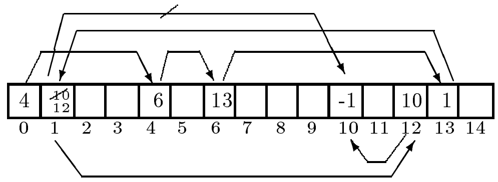


Im allgemeinen Fall jedoch liegt ein Zyklus vor, der vom letzten Element zu einem früheren Element führt, das nicht notwendig das erste Element ist. Es ist vom ersten verschieden, so *kommt in ptr ein Wert zweimal vor*.

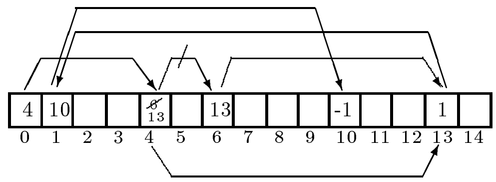


Operationen *insert* und *delete*:

- Einfügen eines Wertes an irgendeiner Stelle in der Liste ist wie gewöhnlich in Zeit $O(n)$ möglich:



- Löschen irgendeines Wertes in der Liste kann auch in Zeit $O(n)$ realisiert werden:



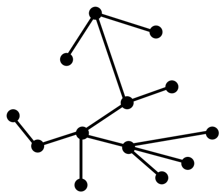
Anmerkung: Es gilt sogar Zeit $O(1)$, falls bereits der Index i in *ptr* gegeben ist, für den gilt: $ptr[i]$ ist der zu verbiegende Pointer.

1.3 Bäume

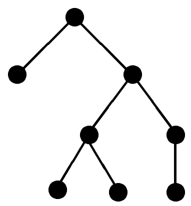
1.3.1 Grundlagen

Definition: Ein *Baum* ist ein zusammenhängender azyklischer Graph. Wir unterscheiden zwei Typen von Bäumen:

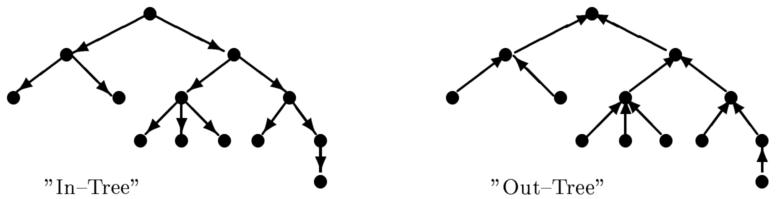
- **ungerichtete Bäume:** Hierbei ist nicht notwendig eine *Wurzel* ausgezeichnet. Nur die *Blätter* sind als solche identifizierbar, denn sie sind die einzigen Knoten, die nur einen Nachbarn haben — im Gegensatz zu den *inneren Knoten*.



  Der *Grad eines Knotens* ist die Anzahl seiner inzidenten Kanten bzw. seiner Nachbarn.
  Übliche Darstellung:



- **gewurzelte Bäume:** Bei ausgezeichneter Wurzel sind dies Graphen der folgenden Gestalt:



  Der *Eingangsgrad* eines Knotens ist die Anzahl der Pfeile zum Knoten, sein *Ausgangsgrad* ist die Anzahl der vom Knoten ausgehenden Pfeile. Der *Grad* eines Knotens ist die Summe seines Eingangs– und seines Ausgangsgrads.

  Beim In–Tree hat jeder Knoten Ausgangsgad 1, beim Out–Tree hat jeder Knoten Eingangsgrad 1, mit Ausnahme der Wurzel.

Satz: Ein ungerichteter Graph $G = (V, E)$ ist ein Baum genau dann wenn er zusammenhängend ist und $|V| = (|E| + 1)$.

Beweis: "$\Leftarrow$": Durch Induktion über $|V|$ zeigt man: Ein zusammenhängender Graph mit $|V| = (|E| + 1)$ ist zyklenfrei und somit nach Definition ein Baum. Beim Induktionsschritt berücksichtigt man, daß die neu hinzukommende Kante zum ebenfalls neu hinzukommenden Knoten inzident sein muß und daher keinen Zyklus erzeugen kann, weil der neue Knoten sonst eine eigene Zusammenhangskomponente bilden würde.

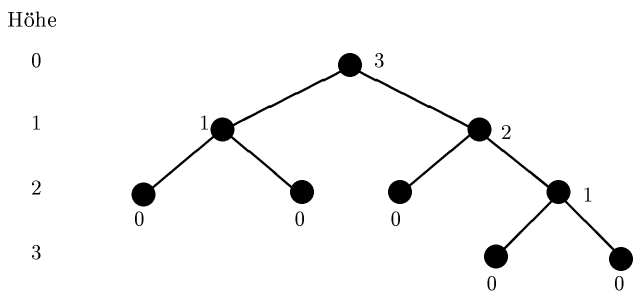
"$\Rightarrow$": Für einen zusammenhängenden azyklischen Graphen gilt $|E| \geq (|V|-1)$, denn für $|E| < (|V|-1)$ ergeben sich mindestens zwei Zusammenhangskomponenten; Für $|E| \geq |V|$ hätte jeder Knoten mindestens eine Kante zugeordnet. Notwendigerweise würde ein Zyklus vorliegen. Insgesamt folgt: $|E| = |V| - 1$, *q.e.d.*

Satz: Ein ungerichteter Graph ist genau dann ein Baum, wenn zwischen je zwei Knoten x, y mit $x \neq y$ genau ein Pfad existiert.

Beweis: "$\Rightarrow$": Ein Baum ist ein zusammenhängender azyklischer Graph. Wegen der ersten Eigenschaft existert stets ein Pfad zwischen zwei verschiedenen Knoten, wegen der zweiten Eigenschaft ist er eindeutig, denn sonst wäre die Vereinigung beider Pfade (bei ungerichteter Sichtweise) ein Zyklus.
"$\Leftarrow$": Zwischen je zwei Knoten existiere je genau ein Pfad. Dann existiert stets mindestens einer, also ist der Graph zusammenhängend. Läge ein Zyklus vor, so gäbe es ein Knotenpaar mit mehr als einem verbindenden Pfad. Deshalb ist der Graph kreisfrei, also ein Baum, *q.e.d.*

Definitionen (Baum–Terminologie): Bei Definition einer Wurzel im Baum und bei Betrachtung des Baums als Out–Tree werden die Begriffe *Vater (unmittelbarer Vorfahr)* und *Sohn (unmittelbarer Nachfahr)* sinnvoll. Man erhält dadurch:

- *Vorfahr eines Knotens*: Vater oder Vorfahr des Vaters
- *Nachfahr eines Knotens*: Sohn oder Nachfahr des Sohns
- *Geschwister eines Knotens*: Sohn desselben unmittelbaren Vorfahren
- *Tiefe eines Knotens*: Länge des eindeutig bestimmten Pfades von der Wurzel zum Knoten
- *Länge eines Pfades*: Anzahl der Kanten auf dem Pfad
- *Höhe einse Knotens*: Tiefe seines Unterbaums
- *Tiefe eines Baums*: Maximale Tiefe eines Knotens (also eines Blattes)
- *Höhe eines Baums*: seine Tiefe

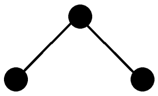


- *Niedrigster gemeinsamer Vorfahr* zweier Knoten: kanonisch
- *Wald*: Ein Graph, dessen Zusammenhangskomponenten sämtlich Bäume sind

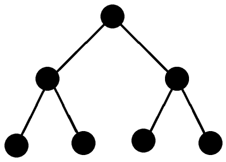
1.3.2 Binärbäume

Definition: Ein *Binärbaum* ist ein gerichteter oder ungerichteter Baum, bei dem jeder Knoten höchstens zwei Kinder hat.

- Ein Binärbaum heißt *geordnet* genau dann wenn jeder (innere) Knoten ein erstes und eventuell ein zweites Kind hat. Allgemeiner kann man geordnete Bäume beliebigen Verzweigungsgrads erklären.
- Ein Binärbaum heißt *voll* genau dann wenn jeder Knoten entweder 0 oder 2 Kinder hat.
- Ein Binärbaum heißt *vollständig* genau dann wenn alle Blätter maximale Tiefe haben und die maximale Anzahl von Blättern vorliegt. Alle Ebenen sind dann maximal ("vollständig") aufgefüllt. Der vollständige Binärbaum der Tiefe n wird als B_n bezeichnet.



B1



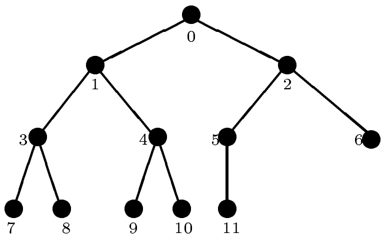
B2

Bemerkung zu B_n: Der vollständige Binärbaum B_n der Tiefe n hat
– insgesamt $\sum_{i=0}^{n} 2^i = \frac{2^{n+1}-1}{2-1} = 2^{n+1} - 1$ Knoten
– 2^i Knoten in Tiefe i
– $(2^{n+1} - 1) - 2^n = 2^n - 1$ innere Knoten
– 2^n Blätter

Linearisierung von Binärbäumen: Für das Durchlaufen eines Binärbaums, d.h. seine Linearisierung, gibt es verschiedene Möglichkeiten:

$$\left\{ \begin{array}{ll} \textbf{pre–order}: & \mathrm{W} — \mathrm{lUb} — \mathrm{rUb} \\ \textbf{in–order}: & \mathrm{lUb} — \mathrm{W} — \mathrm{rUb} \\ \textbf{post–order}: & \mathrm{lUb} — \mathrm{rUb} — \mathrm{W} \end{array} \right.$$

Dabei ist lUb der linke Unterbaum, W die Wurzel und rUb der rechte Unteraum. In dieser Reihenfolge wird rekursiv vorgegangen:



pre–order:	0,1,3,7,8,4,9,10,2,5,11,6
in–order:	7,3,8,1,9,4,10,0,11,5,2,6
post–order:	7,8,3,9,10,4,1,11,5,6,2,0

Satz: Es gibt einen Algorithmus, der zu zwei gegebenen Permutationen einer Menge von Knoten entscheidet[1], ob diese Permutationen pre– und post–order eines Binärbaums sind und dabei ggf. den zugehörigen Binärbaum konstruiert.

[1] In der Terminologie der Entscheidbarkeitstheorie bedeutet dies die Rekursivität der Sprache aller Paare von Permutationen von V, deren erstes Element die pre–order–Linearisierung und deren zweites Element die post–order–Linearisierung eines Binärbaums ist.

Beweis: Gegeben seien die beiden Permutationen als Listen. Grundidee: Anhand dieser beiden Permutationen kann jeweils festgestellt werden, welcher Knoten die Wurzel des gerade betrachteten Baums ist, und an welcher Stelle die Grenze zwischen $\mathbb{lUb}$ und $\mathbb{rUb}$ liegt. Es sind nämlich die beiden Unterbäume in jeder der zwei Listen zusammenhängende Teilsequenzen. Man spalte die Wurzel ab, bestimme in der post– bzw. pre–order–Sequenz die Grenze zwischen $\mathbb{lUb}$ und $\mathbb{rUb}$ und fahre rekursiv mit dem $\mathbb{lUb}$ und dann mit dem $\mathbb{rUb}$ fort.
Beispiel:

$$\overbrace{\ 0\ }^{\mathbb{W}},\overbrace{\mathbf{1},3,7,8,4,9,10}^{\mathbb{lUb}},\overbrace{2,5,11,6}^{\mathbb{rUb}}$$

$$\underbrace{7,8,3,9,10,4,\mathbf{1}}_{\mathbb{lUb}},\underbrace{11,5,6,2}_{\mathbb{rUb}},\underbrace{\ 0\ }_{\mathbb{W}}$$

- Invariante der Rekursion: Das erste Element der pre–order–Liste muß gleich dem letzten Element der post–order–Liste sein, das ist die Wurzel.

- Das *zweite* Element der pre–order–Liste gibt das Element an, bei dem in der *post–order–Liste* die Trennung zwischen $\mathbb{lUb}$ und $\mathbb{rUb}$ liegt. Damit erhält man auch die Information, an welcher Stelle in der pre–order–Liste zu trennen ist. Begründung: Der zweite Knoten in der pre–order–Liste (im Bsp: 1) ist die Wurzel des $\mathbb{lUb}$. Gleichzeitig ist er der letzte in post–order–Reihenfolge besuchte Knoten des $\mathbb{lUb}$.

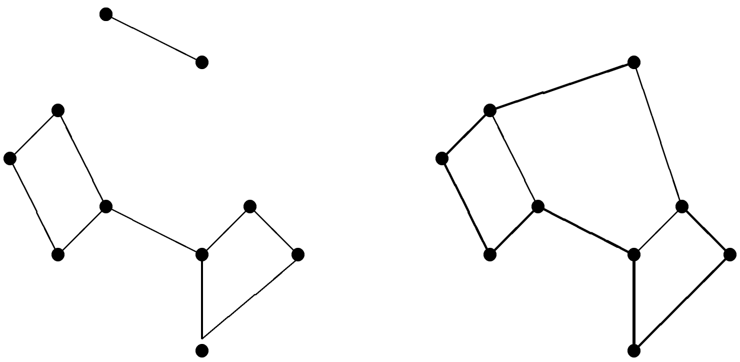
Algorithmus:
$< a >$ sei die pre–order–Sequenz, $< b >$ sei die post–order–Sequenz.

```
test(<a>,<b>):    if (<a>=<b>=leer) then TRUE
                  elsif (genau eine der beiden =leer) then FALSE
                  else
                   w:= 1. Knoten in <a>;
                   if ((letzter Knoten in <b>)#w) then FALSE fi;
                   v:= 2. Knoten in <a>;
                   <a>:=<a> ohne 1. Knoten;
                   <b>:=<b> ohne letzten Knoten;
                   rUb<b>:= Teilfolge von <b> aller Knoten rechts von v;
                   rUb<a>:= Teilfolge von <a> aller Knoten in rUb<b>;
                   lUb<b>:= restliche Teilfolge in <b>;
                   lUb<a>:= restliche Teilfolge in <a>;
                   test(lUb<a>,lUb<b>);
                   test(rUb<a>,rUb<b>)
                  fi;

main():           test(<a>,<b>);
```

Terminierung dieses Algorithmus ist klar, denn alle Sequenzen verkürzen sich echt; Korrektheit folgt aus dem bisher Gesagten. Außerdem ist es einfach, den Algorithmus so zu erweitern daß er zugleich den dargestellten Binärbaum aufbaut. *q.e.d.*

Definition (Spannbaum): Sei $G = (V, E)$ ein zusammenhängender Graph. Ein *Spannbaum* von G ist ein Baum $T = (V, E')$ mit $E' \subset E$. Das heißt, ein Spannbaum verbindet alle Knoten des Graphen und verwendet dazu nur Kanten des Graphen.

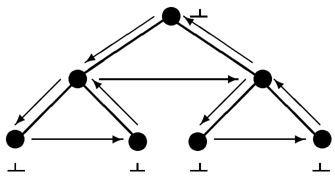


Der erste Graph besitzt keinen Spannbaum weil er nicht zusammenhängend ist, für den zweiten ist ein Spannbaum eingezeichnet. Ist der Spannbaum nur ein einzelner Pfad, so spricht man von einem *Hamilton–Pfad*.

1.3.3 Implementierung von Bäumen

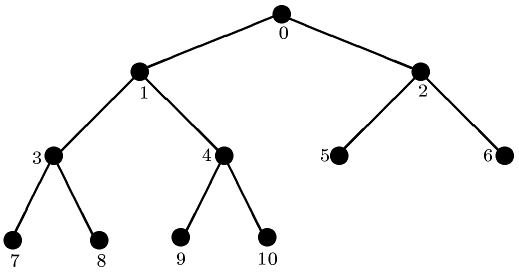
1. als Geflecht mit Pointern *zu* den Kindern (Out–Tree) bzw. *von* den Kindern zu deren unmittelbaren Vorfahren (In–Tree)

2. als Geflecht mit Zeigern zum eigenen ersten Kind und zum eigenen nächsten Geschwister; Im Falle eines letzten Kindes wird stattdessen ein Pointer auf den Vater gerichtet. Für Blätter und die Wurzel werden NIL–Pointer benutzt.



3. als **Vollständig linksbalancierter Binärbaum:** Der Binärbaum wird ebenenweise von links nach rechts aufgefüllt. Werden die Knoten entsprechend ebenenweise ab null numeriert, so ergibt sich:

– Die Kinder des inneren Knoten i sind $(2i+1)$ und $(2i+2)$
– Wurzel: 0
– Vater des Knotens i: $\lfloor \frac{i-1}{2} \rfloor$



Bemerkung: Die Möglichkeiten 1) und 2) sind auf allgemeine Bäume anwendbar, Möglichkeit 3) nur auf Binärbäume.

Kapitel 3

Höhere Datenstrukturen

3.1 Grundlagen

Die grundlegenden Operationen: Viele Algorithmen arbeiten mit Daten, die als Mengen organisiert sind. Folgende Liste enthält einige Grundoperationen, die durch geeignet gewählte Datenstrukturen implementiert werden können.

iselement(x,S)	liefert $(x \in S) \in \mathbb{B}$
insert(x,S)	$S := S \cup \{x\}$
delete(x,S)	$S := S \backslash \{x\}$
deletemin(S)	$S := S \backslash \{\min(S)\}$, falls S in geordnetem Universum
replace(x,x',S)	$S := S \backslash \{x\} \cup \{x'\}$
decrease-key(x,Δ,S)	Ersetze Element $x = (k, v)$ durch $x' = (k - \Delta, v)$
union(S_i,S_j)	$S_i := S_i \cup S_j$
merge(S_i,S_j)	*union(S_i,S_j)*, falls S_i, S_j disjunkt
find(x)	Falls $x \in \bigcup\limits_i S_i$, so finde j mit $x \in S_j$, sonst undefiniert
findmin(S)	*find*(min(S)), falls geordnetes Universum vorliegt
findmax(S)	*find*(max(S)), falls geordnetes Universum vorliegt
split(S_i,x,S_j)	Falls $x \in S_i$: $\begin{cases} S_j := \{b \in S_i \mid key(b) > key(x)\} \\ S_i := \{b \in S_i \mid key(b) \leq key(x)\} \end{cases}$
concatenate(S_i,S_j)	Falls $key(findmax(S_i)) < key(findmin(S_j))$, dann: $S_i := S_i \cup S_j$

Abstrakte Datenstrukturen:

1. *"Wörterbuch"("dictionary")*
   - Operationen: *iselement, insert, delete*
   - Implementierung als: hash–Tabellen, balancierte Bäume
2. *"Vorrangwarteschlange"("priority queue")*
   - Operationen: *(iselement)*[1]*, insert, delete, find-min*
   - Implementierung als balancierte Bäume, Linksbäume ("leftist–trees"), Binomialwälder ("binomial forests")
3. Spezialfall: *"Mergeable heap"*
   - Operationen: *(iselement), insert, delete, find-min, merge*
   - Implementierung als (2,3)–Bäume, Binomialbäume, leftist–trees
4. *concatenate queue*
   - Operationen: *insert, delete, split, merge*
   - Implementierung als (2,3)–Bäume

3.2 Allgemeine Suchbäume — balancierte Bäume

Definition: Bei einem *externen Suchbaum* werden die Datenelemente nur an den Blättern gespeichert und die inneren Knoten enthalten Verwaltungsinformationen (siehe z.B. (a,b)–Bäume). Bei *internen Suchbäumen* liegen die Datenelemente an den internen Knoten. Zeiger auf Blätter sind NIL–Pointer. Daher kann man auch sagen: An *allen* Knoten sind Daten gespeichert.

3.2.1 Höhenbalancierte Bäume

Anmerkung: Liegen die betrachteten Mengen nicht in einem geordneten Universum U, so müssen beim Suchen alle Elemente betrachtet werden. Ist das Universum aber total[2] geordnet, so ist ein Test nicht nur auf Gleichheit möglich. Dann ist für die Suche nach einem bestimmten Element das *divide & conquer*–Verfahren anwendbar (entsprechend binärer Suche).

(a,b)–Bäume

Definition: Ein *(a,b)–Baum* ist ein externer Suchbaum, bei dem alle Blätter dieselbe Tiefe haben, und für den gilt:

$a \leq \#$ Kinder $\leq b$ an allen Knoten außer der Wurzel
$2 \leq \#$ Kinder $\leq b$ an der Wurzel
$b \geq (2a-1)$

Für den Spezialfall $a = 2, b = 3$ ergeben sich die sog. *(2,3)–Bäume*, und im Falle $b = 2a-1$ spricht man von *B–Bäumen*. Die von (a,b)–Bäumen unterstützten Operationen sind *iselement*, *insert* und *delete*.

Beispiel: Ein (2,4)–Baum

[1] Priority queues sind für Suchoperationen ungeeignet. Es wird i.a. eine andere Datenstruktur parallel gehalten.

[2] Bei totaler Ordnung ist jedes Element mit jedem auf $<,=,>$ vergleichbar, bei partieller Ordnung gilt dies nur für $=$. Der Test auf $<,>$ ist nur zwischen bestimmten Elementen möglich.

2 a

1 3,4,6

x y

1 2 3 4 6 7

Bemerkung (Bedeutung der Eintragungen an den inneren Knoten): Für den inneren Knoten v sei $d := \mathcal{S}(v) :=\#$ Kinder von v. Dann werden am Knoten v folgende Verwaltungsinformationen eingetragen: $(d+1)$ Schlüsses K_i mit:

- $K_{i-1} <$ alle Schlüssel im i–ten Unterbaum von $v \leq K_i$
- $K_0 = -\infty$
- $K_d = \infty$

Das bedeutet, daß in der rechten Figur gelten muß: $x \leq a < y$. Diese Informationen ermöglichen die Anwendung des divide & conquer – Schemas für die Operation *iselement*.

Satz: Sei T ein (a,b)–Baum mit n Blättern und Höhe h. Dann gilt:

(i) $2 \cdot a^{h-1} \leq n \leq b^h$

(ii) $\log_b(n) \leq h \leq 1 + \log_a(\frac{n}{2})$

Beweis: An jedem inneren Knoten, der nicht die Wurzel von T ist, gilt für jeden Unterbaum: $a^h \leq n \leq b^h$, wobei h die Höhe des Unterbaums ist und n die Anzahl seiner Blätter. Denn: für $h = 0$ ist $n = 1 = a^0 = b^0$. Für den Induktionsschritt habe nun v mindestens a und höchstens b Kinder, und für den Unterbaum eines jeden gelte, daß er zwischen a^{h-1} und b^{h-1} Blätter habe. Dann hat der Unterbaum an v zwischen $a \cdot a^{h-1}$ und $b \cdot b^{h-1}$ Blätter. Also gilt obige Behauptung.
An der Wurzel liegen mindestens 2 und höchstens b Kinder vor, und deren Unterbäume haben die Höhe $h-1$, also je zwischen a^{h-1} und b^{h-1} Blätter. Somit liegt die Zahl der Blätter zwischen $2 \cdot a^{h-1}$ und $b \cdot b^{h-1} = b^h$, und das ist die Behauptung *(i)*. Für *(ii)* logarithmiere man jeweils die beiden Teilungleichungen, *q.e.d.*

Operationen auf (a,b)–Bäumen:

iselement (x): Es werden die an den internen Knoten abgelegten Verwaltungsinformationen verwendet, für ein Verfahren analog der binären Suche.

```
v:=Wurzel von T;
while (v nicht Blatt) do
 i:=min({j|1<=j<=S(v) und x<=Kj}); // S(v)=Kinderzahl
 v:=i-tes Kind von v
od;
return (x=key(v))
```

Bemerkung: Da, wie oben gezeigt, ein (a,b)–Baum logarithmische Höhe hat und die Zahl der Schritte zur Minimumsbildung durch $b = O(1)$ beschränkt ist, ist *iselment* in Zeit $O(\log(n))$ ausführbar, wobei natürlich a und b konstant sind.

insert(x): Das Vorgehen für das Einfügen eines neuen Blattes ist folgendes:

(a) Führe *iselement(x)* aus. Diese Suche führt zu einem Blatt w. Ist x nicht bereits in T enthalten, so ist an diesem Blatt das kleinste Datenelement gespeichert, das größer ist als x.

(b) In diesem Falle füge x links vom gefundenen Blatt w ein. Hierdurch erhöht sich der Verzweigungsgrad $\mathcal{S}(v)$ des gemeinsamen Vaters v von x und w.

(c) Wird hierdurch $\mathcal{S}(v) > b$, so führe folgende Rebalancierung aus:

`v:=gemeinsamer Vater von` x `und` w`;`
`while` $\mathcal{S}(v) > b$ `do`
 `if (`$v \neq$`Wurzel) then` u`:=Vater von` v
 `else` u`:= neue Wurzel mit einzigem Kind` v `fi;`
 `Zerlege die Menge` G `aller Kinder von` v `in zwei gleichgroße`
 `Teilmengen` G_1 `und` G_2`, so daß` $a \leq |G_1|, |G_2| \leq b$`;`
 `'Zerteile' Vater` v `in zwei Väter` v_1`,`v_2 `und mache` v_1
 `zu Vater von` G_1 `und` v_2 `zu Vater von` G_2`; (*)`
 v`:=`u`; (**)`
`od`

Bemerkung: `(*)` Aus dem Sohn v von u werden zwei Söhne v_1 und v_2, also erhöht sich der Grad $\mathcal{S}(u)$ um eins. `(**)` Wird er größer als b, also gleich $(b+1)$, so ist ein analoges Vorgehen an u nötig, u.s.w. Dies wird durch die Anweisung `v:=u` und die `while`–Schleife bewirkt. Die Verwaltungsinformationen an den behandelten Knoten sind jeweils neu zu berechnen.

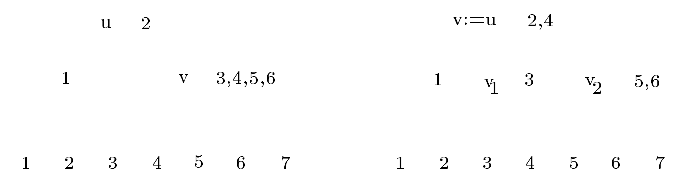
Korrektheit der Teilung: Eine Teilung der Kindermenge eines Knotens und seine Aufspaltung in zwei Knoten ist notwendig, wenn die Kinderzahl den Wert $(b+1)$ erreicht. Durch die Teilung wird $|G_1| = \lceil \frac{b+1}{2} \rceil$ und $|G_2| = \lfloor \frac{b+1}{2} \rfloor$. Liegen diese nun im zulässigen Bereich zwischen a und b? Es gilt:

$\lceil \frac{b+1}{2} \rceil \leq b \Leftrightarrow \frac{b+1}{2} \leq b \Leftrightarrow b \geq 1$

$\lfloor \frac{b+1}{2} \rfloor \geq a \Leftrightarrow \frac{b+1}{2} \geq a \Leftrightarrow b \geq 2a+1$

Letzteres gilt nach Voraussetzung.

Beispiel:



Bemerkung: Der Zeitaufwand für *insert* ist $O(\log(n))$, wobei n die Anzahl der Blätter des Baums T ist[3].

Beweis: Die Operation *iselement* (Laufen zu einem Blatt) kostst Zeit $O(\log(n))$. Eine Teilungsoperation muß auf jeder Ebene des Baums $\leq 1\times$ ausgeführt werden und kostet jedesmal konstante Zeit. Da die Höhe des Baums $O(\log(n))$ ist, folgt die Behauptung, *q.e.d.*

delete(x): Analog zu *insert* erfordert *delete* auch ein Vorgehen in drei Schritten:

[3] Zur Erinnerung: Die Blätter haben alle dieselbe Tiefe.

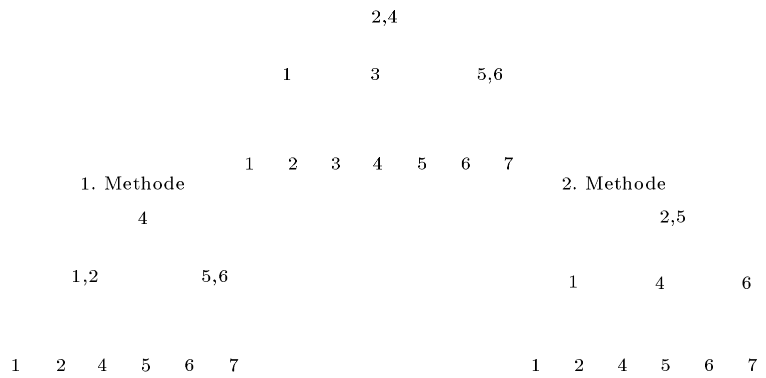
(a) Führe *iselement* aus und laufe dadurch zu einem Blatt.

(b) Falls das Blatt das Element x enthält, so lösche es. Dadurch verkleinert sich der Grad $\mathcal{S}(v)$ des Vaters v.

(c) Wird $\mathcal{S}(v) < a$, so wird Rebalancierung nötig. Für diese gibt es, wenn man einen bestimmten Nachbarn von v betrachtet, zwei Fälle:
(i) Falls der Nachbar von v den Grad $\mathcal{S} \leq a$ hat, so verschmelze v mit diesem Nachbarn.
(ii) Falls der Nachbar von v den Grad $\mathcal{S} > a$ hat, so soll v ein Kind von seinem Nachbarn adoptieren.

Bemerkung: Im Fall *(i)* wird der Grad des Vaters der beiden verschmolzenen Knoten kleiner, und eventuell $< a$, so daß sich die Verletzung der Balancierungsbedingung nach oben fortfplanzen kann. Im Fall *(ii)* gilt dies sicher nicht. Da nun v i.a. zwei Nachbarn hat, hat man in manchen Fällen die Wahl zwischen beiden Methoden. In diesem Fall wählt man natürlich Methode *(ii)*.

Beispiel: Löschen von 3



Satz: Für (a,b)–Bäume benötigen alle Operationen, *iselement, insert, delete* die Zeit $O(\log(n))$.

Bemerkung (Wahl von a):

- Wird der (a,b)–Baum im Hauptspeicher gehalten, so wähle *a klein*, dann sind die Kosten für das Suchen minimal.

- Ist er dagagen im Sekundärspeicher abgelegt, wo die Zugriffszeit für das erste byte eines Zugriffs sehr groß und für die restlichen bytes klein ist, so wähle *a groß*, so daß möglichst wenig erste Zugriffe stattfinden.

Bemerkung (Wahl von b): Voraussetzung ist $b \geq 2a-1$. Wählt man sogar $b \geq 2a$, so sinken die durchschnittlichen Rebalancierungskosten. Die amortisierten Kosten bleiben aber für alle Operationen konstant.

Top–Down–Balancierung: Die Balancierung erfolgt im Herabsteigen von der Wurzel. Dabei wird für den je aktiven Knoten sichergestellt, daß sein Grad $\mathcal{S}$ zwischen a und b liegt. Wird an einem Knoten Rebalancierung erforderlich, so wird eine der oben dargestellten Methoden angewendet.

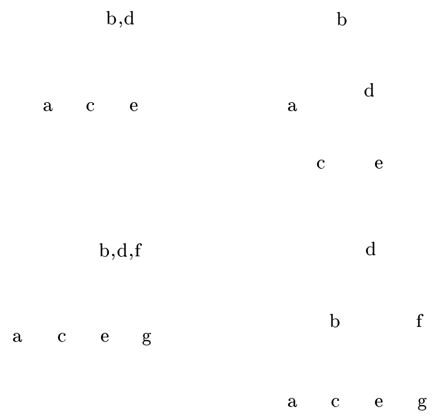
3.2.2 Rot–Schwarz–Bäume zur Implementierung von (a,b)–Bäumen

Definition: Ein *Rot–Schwarz–Baum* ist ein Binärbaum mit roten und schwarzen Kanten und den Eigenschaften

- Alle Blätter haben schwarze Eingangskanten
- Alle Blätter haben dieselbe Schwarztiefe, wenn auch evtl. nicht dieselbe Tiefe
- Auf keinem Pfad gibt es zwei aufeinanderfolgende rote Kanten

Dabei ist die *Schwarztiefe* eines Knotens die Anzahl der schwarzen Kanten auf dem Pfad von der Wurzel zu diesem Knoten.

Implementierung eines (2,4)–Baums als Rot–Schwarz–Baum:



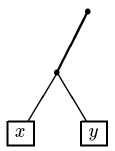
Bemerkung: Die Schwarztiefe aller Blätter ist gleich, und sie ist gleich der ursprünglichen Tiefe der Blätter im (a,b)–Baum. Zur Erinnerung: Im (a,b)–Baum haben alle Blätter gleiche Tiefe.

Operationen auf Rot–Schwarz–implementierten (a,b)–Bäumen:

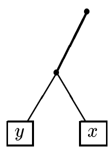
insert(x): Zum Einfügen eines Elements x sind wieder 3 Schritte nötig:

(a) Führe *iselement(x)* aus, was zu einem Blatt $\boxed{y}$ führt. Sei dort noch nicht das Element x gespeichert.

(b) Die Kante zu y ist notwendig schwarz. Es wird wie folgt umgebaut:
Falls $x < y$:



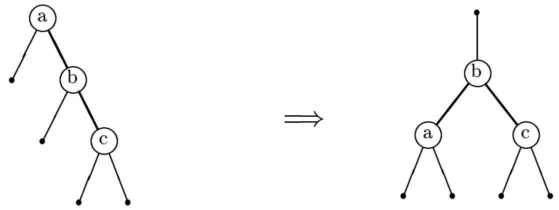
Falls $y < x$:



Die Kante zum früheren y–Blatt (s.o.) wird umgefärbt und ist nun rot.

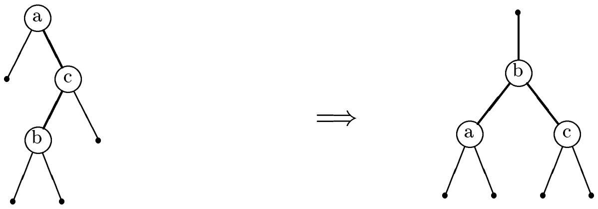
(c) Hierdurch können sich zwei aufeinanderfolgende rote Kanten ergeben, die eine Rebalancierung nötig machen. Fall 1 repräsentiert zwei aufeinanderfolgende linke bzw. rechte rote Kanten, Fall 2 zwei aufeinanderfolgende rote Kanten an verschiedenen Seiten.

1. Fall:



"Einfache Rotation"

2. Fall:

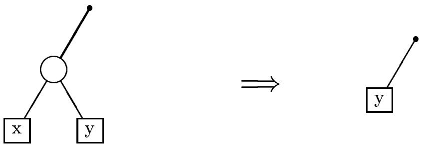


"Doppelte Rotation"

Man beachte: Entgegen der obingen Darstellung handelt es sich bei Rot–Schwarz–Bäumen um externe Suchbäume.

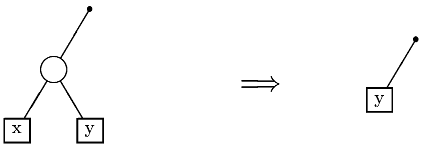
delete(x): Das Löschen des Blattes x — es erfolgt wieder in drei Schritten — macht es erforderlich, daß der Vater des Blattes durch den Bruder des Blattes ersetzt wird, damit die Binärbaumstruktur erhalten bleibt. Der Bruder von x ist natürlich eindeutig bestimmt. Dabei treten zwei Fälle auf:

1. Fall:



Dadurch ändert sich die Schwarztiefe von y nicht, denn die Eingangskante zu y wird (notwendig) zu schwarz umgefärbt. Auch kann es hierdurch nicht zu zwei aufeinanderfolgenden roten Kanten kommen.

2. Fall:



Hierdurch verkleinert sich die Schwarztiefe von y um 1. Man kann sie nicht wieder erhöhen, ohne dabei auch die Schwarztiefe anderer Blätter zu vergrößern. Daher muß sie für alle anderen Blätter des Baums um 1 verkleinert werden. Man betrachtet dazu den Pfad von y zur Wurzel als unveränderlich und färbt Kanten so um, daß auf allen anderen Pfaden jeweils genau eine Kante von schwarz zu rot wird. Ferner darf dabei nicht die Situation entstehen, daß zwei aufeinanderfolgende Kanten rot werden. Ist dies nicht vermeidbar, so muß gemäß obigem Schema rebalanciert werden.

Satz: In Rot–Schwarz–implementierten (a,b)–Bäumen können die Operationen *iselement, insert, delete, minimum, deletemin* in Zeit $O(\log(n))$ implementiert werden.

3.3 Binäre Suchbäume

Gegeben sei das Universum U von Datenelementen mit der totalen Ordnung $<$ auf U. Zur Datenstruktur der Suchbäume werden im allgemeinen folgende Operationen definiert:

- *iselement*
- *insert*
- *delete*

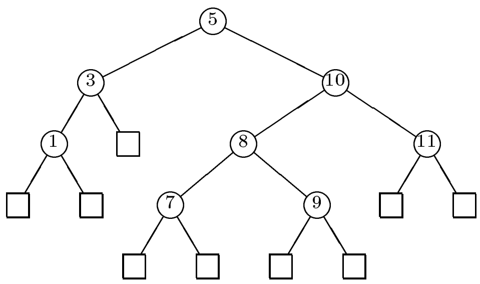
Definition: Bei einem *binären Suchbaum* werden die Datenelemente in den inneren Knoten gespeichert, die Zeiger auf Blätter sind NIL–Pointer. Ferner gilt folgende Invariante:

$$\boxed{\text{Knoten im } \mathbb{lUb} \leq key(\text{Wurzel}) \leq \text{Knoten im } \mathbb{rUb}}$$

Satz: Wird ein binärer Suchbaum in in–order–Reihenfolge linearisiert, so ergibt sich eine aufsteigende Folge.

Beweis: Strukturelle Induktion: Nach Induktionsvoraussetzung gilt die Behauptung je für den $\mathbb{lUb}$ und den $\mathbb{rUb}$. Und nach Voraussetzung gilt: Knoten im $\mathbb{lUb} \leq key(\text{Wurzel}) \leq$ Knoten im $\mathbb{rUb}$. Bei der in–order–Linearisierung werden diese Sequenzen in angegebener Reihenfolge konkateniert und bilden so eine aufsteigende Folge, *q.e.d.*

Beispiel eines binären Suchbaums:



3.4 Natürliche Suchbäume:

Für binäre Suchbäume gilt natürlich insbesondere die Invariante "Knoten im lUb $\leq$ *key*(Wurzel) $\leq$ Knoten im rUb". Sie entstehen durch sukzessive Anwendung von *insert* auf den anfangs leeren Baum.

Operationen:

iselement:

```
iselement(x,r): if (key(r)=x) then return true
                elsif (r=nil) then return false
                elsif (x<key(r)) then return iselement(x,r.lUb)
                else return iselement(x,r.rUb)
                fi
```

$\boxed{\text{Zeitkomplexität: } O(\log(n)) \text{ bis } O(n).}$

insert: Beim Einfügen eines neuen Elements wird dieses jedenfalls ein Blatt.

```
insert(x,r):    if not iselement(x,r) then
                 if (r=nil) then <h"ange x als Blatt an>
                 elsif (x<key(r)) then insert(x,r.lUb)
                 else insert(x,r.rUb)
                 fi
                fi
```

$\boxed{\text{Zeitkomplexität: } O(\log(n)) \text{ bis } O(n).}$

delete: Wenn das Element im Suchbaum enthalten und nicht zufällig ein Blatt ist, so ist das zu löschende Element durch ein anderes zu ersetzen, so daß die Binärbaumstruktur erhalten bleibt. Damit auch die Invariante "Knoten im lUb $\leq$ *key*(Wurzel) $\leq$ Knoten im rUb" bestehen bleibt, wird wie folgt vorgegangen: Nimm das in–order–nächste oder das in–order–vorhergehende Element, d.h. das äußerste linke Blatt im rUb oder das äußerste rechte Blatt im lUb als Ersetzungselement und lösche es.

$\boxed{\text{Zeitkomplexität: } O(\log(n)) \text{ bis } O(n).}$

Problematik der Entartung natürlicher Suchbäume: Werden Elemente in *aufsteigender oder absteigender Reihenfolge* eingefügt, so liefert obige *insert*–Operation einen linear entarteten Suchbaum, nämlich eine Liste. Diese läßt das Suchen (*iselement*) natürlich nur in Zeit $O(n)$ zu, nicht mit $O(\log(n))$. Gleiches gilt für die Operationen *insert* und *delete*.

Satz: Entsteht ein natürlicher Suchbaum durch *insert*–Operationen mit Datenelementen in zufälliger Reihenfolge, so erhält man für alle Operationen die Zeitkomplexität $O(\log(n))$.
Sind dagegen topologisch gleichaussehende Bäume gleichwahrscheinlich, so ergibt sich die Komplexität $\Theta(\sqrt{n})$.

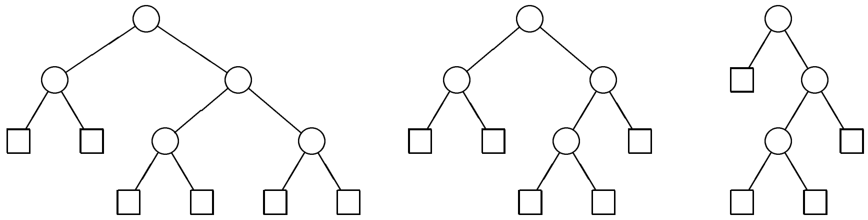
3.5 Balancierte binäre Suchbäume:

3.5.1 Höhenbalancierte Suchbäume (AVL–Bäume)

Definition: *AVL–Bäume*[4] sind spezielle Suchbäume, die neben der gewöhnlichen Sortiertheitsbedingung auch eine Balancierungsbedingung erfüllen:

$$\boxed{|\text{Höhe}(\mathtt{lUb}^{u})\text{-Höhe}(\mathtt{rUb}^{u})| \leq 1 \text{ für alle Knoten } u}$$

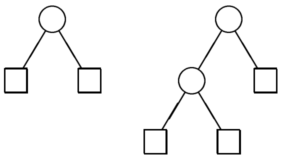
Beispiele: Nur die ersten zwei Bäume sind AVL–Bäume



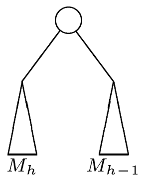
Satz: Die minimale Anzahl M_h von Blättern in einem AVL–Baum der Höhe h ist

$$M_h = fib(h+2)$$

Beweis: Induktion nach h:
I) Es gilt: $M_1 = 2$ und $M_2 = 3$



II) Bei einem bzgl. der Blätterzahl minimalen AVL–Baum der Höhe h darf nur einer der Unterbäume die Höhe $h-1$ haben. Der andere muß kürzer sein, aber wegen der Balancierungsbedingung mindestens Höhe $h-2$ haben. Beide sind natürlich 'minimale' AVL–Bäume. Also gilt: $M_h = M_{h-1} + M_{h-2}$. Dies ist genau die Bestimmungsgleichung für $fib(h+1)$, *q.e.d.*



Folgerung:

1. AVL–Bäume der Höhe h haben mindestens $fib(h+2)$ Blätter, also $fib(h+2)-1$ interne Knoten.

2. Da andererseits gilt: $fib(n) \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$, hat ein AVL–Baum mit n Bättern (oder internen Knoten) die Höhe $O(\log(n))$. Dies ist ein bedeutender Fortschritt gegenüber der bisweilen bei natürlichen Suchbäumen entstehenden Zeitkomplexität von $O(n)$ für die Suchoperation *iselement*.

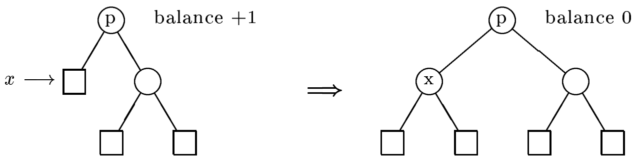
[4]nach Adelson–Velskii–Landis

3.5.2 Operationen auf AVL–Bäumen

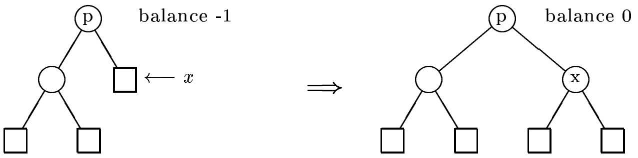
iselement: Diese Operation wird wie bei natürlichen Suchbäumen implementiert. Wie oben ausgeführt, haben aber AVL–Bäume mit n Datenelementen eine Höhe von $O(\log(n))$, woraus logarithmische Zeit für das Suchen folgt.

insert: Zunächst wird eine *iselement*–Operation ausgeführt, die für den Fall, daß das einzufügende Element nicht bereits enthalten ist, zu einem Blatt führt. Dies ist die Stelle, an der das neue Element einzufügen ist. Dabei ergeben sich 3 Fälle:

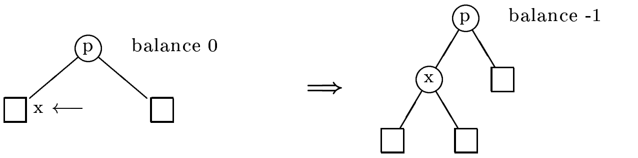
1.Fall:



2.Fall:

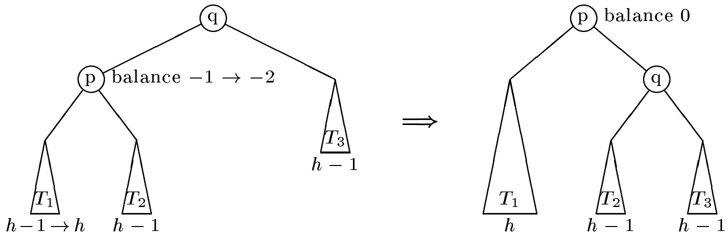


3.Fall:



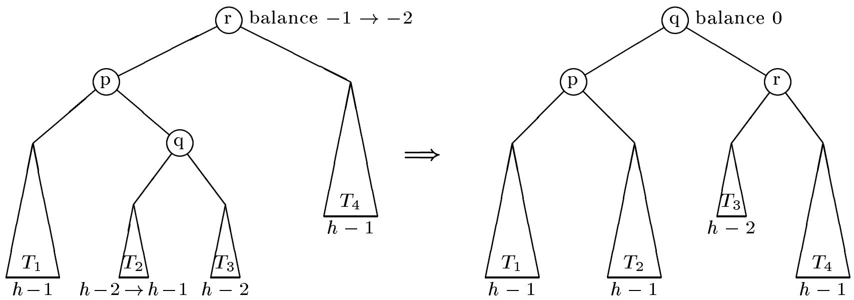
Hier ist eventuell eine Rebalancierung auf dem Pfad zur Wurzel notwendig, denn die Höhe eines Unterbaums ändert sich derart, daß die Balance nicht zu 0 wird, sondern zu $+1$ oder zu -1. Zur Rebalancierung:

Problemfall a.



$(T_1 < p < T_2 < q < T_3)$ **"Einfache Rechtsrotation"**

Problemfall b.



$(T_1 < p < T_2 < q < T_3 < r < T_4)$ **"Doppelte Rechtsrotation"**

delete: Auch diese Operation wird wie bei natürlichen Suchbäumen implementiert. Doch am Ende hat ggf. noch eine Rebalancierung auf dem Pfad vom gelöschten Blatt (das an die Stelle des zu löschenden Elements geschrieben wird) zur Wurzel zu erfolgen.

Satz: Bei AVL–Bäumen sind die Operationen *iselement*, *insert* und *delete* so implementierbar, daß sie die Zeitkomplexität $O(\log(n))$ haben, wobei n die Anzahl der Datenelemente (oder die Anzahl der Blätter) ist.

3.6 Priority Queues (Heaps, Vorrangwarteschlangen)

Für unser Universum U mit totaler Ordung $<$ wird nun eine Datenstruktur mit den

Operationen

- *iselement*
- *insert*
- *delete*
- *deletemin*
- *minimum*
- *decrease-key*
- *merge* (disjunkt)

definiert. Wir betrachten *binomial queues und Fibonacci heaps*[5] als mögliche Implementierungen von priority queues.

3.6.1 Binomial Queues (Binomial Heaps)

Um binomial queues betrachten zu können, definieren wir zunächst ihre nichttrivialen Bestandteile: *Binomialbäume.*

Definition: Die *Binomialbäume* sind rekursiv wie folgt definiert:

B_0 B_1 B_2 .... B_k

B_{k-1}

B_{k-1}

Achtung: Binomialbäume sind keine Binärbäume! Wie in folgender Darstellung dieser Bäume deutlich wird, hat B_n an der Wurzel Grad n.

Satz: Ein Binomialbaum läßt sich wie folgt zerlegen:

(i) B_k

B_0

B_{k-1} B_{k-2} B_{k-3}

(ii) B_k

B_{k-1}

B_{k-2}

B_{k-3}

B_0

B_0

Beweis: Strukturelle Induktion

Satz: Es gilt:

1. B_n hat 2^n Knoten

[5] Vuillemin, Brown

2. Die Wurzel von B_n hat Grad n
3. B_n hat Höhe n
4. B_n hat $\binom{n}{i}$ Knoten in Tiefe i.

Beweis:

zu 1: Induktion unter Verwendung der Definition von Binomialbäumen

zu 2: Siehe Zerlegung (i) für Binomialbäume

zu 3: Siehe Zerlegung (ii) für Binomialbäume und Induktion

zu 4: Induktion über n;
I) $n = 0$: B_0 hat 1 Knoten in Höhe $i = 0$ und 0 Knoten in Höhe $i > 0$, also $\binom{0}{i}$ Knoten in Höhe i.
II) Ist N_i^n die Anzahl der Knoten in Höhe i in B_n, so gilt unter der entsprechenden Induktionsannahme für B_n, daß $N_i^{n+1} = N_i^n + N_{i-1}^n = \binom{n}{i} + \binom{n}{i-1}$, *q.e.d.*

Beispiele:

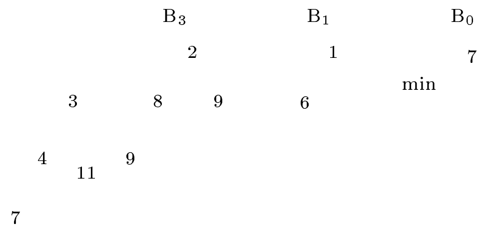
Bemerkung: Eine mögliche Implementierung von Binomialbäumen ist das Schema "Pointer zu 1. Kind und zu nächstem Geschwister".

Verwendung von Binomialbäumen für den Aufbau von Binonial Queues

Definition: Eine *Binomial Queue* mit N Elementen wird wie folgt aufgebaut:

1. Betrachte die Binärdarstellung bin(N) von N
2. Für jedes gesetzte bit ("1") an Position i wird ein Binomialbaum vom Typ B_i benötigt. Dieser hat bekanntlich 2^i Knoten.
3. Verbinde die Binomialbäume in einer doppelt verketteten zirkulären Liste.
4. Beachte, daß innerhalb jedes Binomialbaums die Minimums–Variante der Heap–Bedingung, nämlich $\boxed{\text{Vater} \leq \text{Sohn}}$ erfüllt sein muß. Dadurch ist die Wurzel eines einzelnen Binomialbaums gleichzeitig sein minimales Element.
5. Richte einen Pointer auf das Element in der Wurzel–Liste mit minimalem Schlüssel. Es ist dann das kleinste von allen Elemeneten in der Binomial Queue.

Beispiel: $n = 11 = (1011)_2$
Operationen auf Binomial Queues:



iselement: Die Heap–Bedingung wirkt sich nur auf die Anordnung der Datenelemente innerhalb jedes einzelnen Binomialbaums aus, regelt aber nicht, in welchem Binomialbaum ein gegebenes Element gespeichert sein muß. Tatsächlich kann ein Element in jedem der vorhandenen Binomialbäume stehen. Das Suchen ist hier nicht effizient implementierbar, denn im worst case muß jedes Element eines Binomialbaums angeschaut werden[6]. B_i enthält 2^i Knoten, und eine Binomial Queue mit N Elementen besitzt maximal die Binomialbäume $B_0, B_1, \cdots, B_{\log(N)-1}$. Also wären zum Suchen eines Elements schlimmstenfalls $2^0 + 2^1 + \cdots + 2^{\log(N)-1} = N - 1 = O(N)$ Elemente zu betrachten. Daher wird eine gesonderte Datenstruktur, ein Suchbaum, für die *iselement*–Operation verwendet. Dort ist Suchen mit Zeitaufwand $O(\log(n))$ möglich.

merge: Das Vorgehen für das merge zweier Vorrangwarteschlangen entspricht genau der Addition zweier Binärzahlen: Ein einzelnes B_i wird übernommen, aus zwei B_is wird ein B_{i+1} konstruiert. Damit die Heap–Bedingung erhalten bleibt wird als Wurzel des entstehenden Binomialbaums B_{i+1} die kleinere der Wurzeln der beiden B_i genommen. Allerdings kann ein solcher Übertrag dazu führen, daß im nächsten Verschmelzungsschritt *drei* B_{i+1} zu verschmelzen sind. Dann wird unter Beachtung obiger Regeln ein B_{i+2} gebildet, und der verbleibende B_{i+1} unverändert gelassen.

Algorithmus

```
merge: for k:=0,1,2,... do
        if (es ex. 1 Bk) then "ubernimm dieses
        else if (es ex. 2 Bk) then verbinde 2 Bk's zu einem
                B{k+1} und w"ahle als Wurzel die kleinere der
                beiden Wurzeln
       else if (es ex. 3 Bk) then verbinde 2 davon zu einem
               B{k+1} (wie oben) und lasse den dritten Bk
               unver"andert. ((nur f"ur k>0))
        fi
       od
```

Komplexität: Binomial Queue Nr. 1 mit N_1 Elementen hat $\leq \log(N_1)$ Binomialbäume[7], und analog hat Binomial Queue Nr 2 mit N_2 Elementen $\leq \log(N_2)$ Binomialbäume. Sei $N := N_1 + N_2$. Ein einzelner merge–Vorgang, das Zusammenhängen zweier Binomialbäume, erfolgt in konstanter Zeit. Natürlich muß er

[6] binäres Suchen ist nicht möglich

[7] N hat $\leq \log(N)$ 1en in seiner Binärdarstellung

höchstens so oft ausgeführt werden wie Binomialbäume in der resultierenden Binomial Queue vorliegen. Diese Anzahl wiederum ist höchstens $\log(N)$. Also ist die Zeitkomplexität $O(\log(N)) = O(\log(N_1 + N_2))$.

Bemerkung: Es werde eine binomial queue, beginnend mit der leeren queue, durch fortgesetztes *merge* mit trivalen einelementigen heaps aufgebaut. Dabei möge sowohl das Einfügen dieses neuen einelementigen Baums als auch jeder rekursive Verschmelzungsschritt eine Zeiteinheit kosten, denn jede dieser Operationen ist von der Komplexität $O(1)$. Wir sind an den gesamten Kosten zum Aufbau einer binomial queue mit N Elementen interessiert. Diese sind übrigens identisch zum Aufwand für das binäre Hochzählen von 0 bis N, wenn jeder Zählschritt und jeder Übertrag je eine Zeiteinheit kosten:

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{0+1}_{1}+1}_{2\text{ Schritte}}+1}_{1}+1}_{3}+1}_{1}$$

Sei a_n die Anzahl der Schritte (Einfügen des neuen Elements und anschließende Verschmelzungen) beim mergen eines Knotens zu einer queue mit $(n-1)$ Elementen. Dann gilt für a_n:

n																
1	1															
2	2															
..4	1	3														
..8	1	2	1	4												
..16	1	2	1	3	1	2	1	5								
..32	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1	6

Man erkennt sofort, daß jede Zeile von der Länge 2_k ist, und daß die k-te Zeile alle Elemente der Zeilen $1, 2, \cdots, (k-1)$ enthält, wobei das letzte Element um eins erhöht ist. Daraus ergibt sich folgende Rekursionsgleichung für a_n:

$$a_1 = 1$$
$$a_n = a_{\frac{n}{2}} + 1 \text{ für } n \in 2^{\boldsymbol{N}}$$
$$a_n = a_{n-2^{\lfloor \log(n) \rfloor}} \text{ sonst.}$$

Diese Bestimmungsgleichung ist unmittelbar einsichtig, denn für $n \in 2^{\boldsymbol{N}_0}$ unterscheidet sich die Situation von $n = 2^i$ Knoten nur insofern von der Situation mit $\frac{n}{2} = 2^{i-1}$ Knoten, daß im ersten Fall noch ein Binomialbaum B_{i-1} vorliegt. Das Hinzunehmen eines weiteren Knotens löst in beiden Fällen dieselbe Folge von Verschmelzungsschritten aus, mit der Einschränkung, daß im letzteren Fall der neu entstandene Baum B_{i-1} mit dem vorhandenen zu einem B_i vereinigt wird. Das ist die "+1" in $a_n = a_{\frac{n}{2}} + 1$. Im Falle $n \notin 2^{\boldsymbol{N}_0}$ liegen ähnliche Verhältnisse vor. Zwei queues mit n bzw. $(n - 2^{\lfloor \log(n) \rfloor})$ Knoten unterscheiden sich nur um einen Baum $B_{\lfloor \log(n) \rfloor}$. Gesucht ist nun $S_n := \sum_{k=1}^{n} a_k$. Statt obige Rekursionsgleichungen zu expandieren und eine explizite Darstellung für a_n zu suchen, die dann aufsummiert wird, betrachten wir einfach S_{2^k}. Obiger Tabelle sieht man an, daß $S_{2^k} = 2 \cdot S_{2^{k-1}} + 1$ gilt. Substituiert man S_{2^k} durch T_k, und löst man die entstehende Rekursionsgleichung $T_k = 2 \cdot T_{k-1} + 1$, so ergibt sich $T_k = 2^{k+1} - 1$, also $S_n = 2 \cdot n - 1$. Folglich ist

$$S_N = O(N).$$

minimum: Diese Operation ist trivial ausführbar, denn es ist ein Pointer auf das minimale Element gegeben. Daher ist die Zeitkomplexität auch $O(1)$.

deletemin: Das Minimum ist auf Grund der Heap–Bedingung Wurzel eines Binomialbaums B_i in der Liste. Wird diese gelöscht, so zerfällt der Binomialbaum in i Teilbäume $B_0, B_1, \cdots, B_{i-1}$ (siehe Zerlegung (i) eines Binomialbaums). Diese sind alle zur verbleibenden Queue zu mergen. Eine ähnliche Rechnung wie in obiger Bemerkung zeigt: Der Zeitaufwand hierfür ist $O(\log(N))$.

delete: Sei das zu löschende Element nicht Wurzel eines Binomialbaums (sonst: siehe oben). Es sei enthalten im Binomialbaum B_i. Nach Zerlegung (ii) von Binomialbäumen gilt:

B_k

B_{k-1}

B_{k-2}

B_{k-3}

B_i

B_0

B_0

- Der Teil unterhalb von B_i ist abzuschneiden. Er bildet selbst einen Binomialbaum vom Typ B_i welcher zur verbleibenden Warteschlange zu mergen ist.
- Oberhalb von B_i haben wir einzelne Binomialbäume $B_{i+1}, \cdots, B_{k-1}$ zu betrachten, die auch jeder zur restlichen Binomial Queue gemerged werden.
- Nun wird das unser Element enthaltende B_i betrachtet: Ist das Element nicht Wurzel von B_i, so gehört es zu einem B_j in B_i mit $j < i$.
  - Schneide $B_0, \cdots, B_{j-1}$ ab. Dieses bildet ein B_j, merge es.
  - etc.

Ingesamt wird merge auf folgende Bäume angewendet: $B_0, B_1, \cdots, B_i, \cdots, B_{k-1}$. Man erhält damit als gesamten Zeitaufwand $O(\log(N))$.

insert: Die *insert*–Operation wird einfach durch die *merge*–Operation implementiert. Die Vorrangwarteschlange, die neu hinzukommt, ist trivial. Sie enthält nur einen Binomialbaum B_0.

decrease-key: Die Hintereinanderausführung von *delete* und *insert* erlaubt eine $O(\log(N))$–Implementierung von *decrease-key.* Selbst wenn zu Beginn noch eine Suchoperation ausgeführt wird, um festzustellen, ob das Element tatsächlich enthalten ist, so bleibt diese Komplexität erhalten — vorausgesetzt, daß für *iselement* eine Datenstruktur Suchbaum verwendet wird.

3.6.2 Fibonacci Heaps

Vorbemerkungen:

- Fibonacci heaps stellen eine Erweiterung der binomial queues dar, und eine weitere Möglichkeit zur Implementierung von heaps. Die amortisierten Kosten[8] für die Operationen *delete-min* und *delete* betragen hierbei $O(\log(N))$, die für alle anderen heap–Operationen lediglich $O(1)$. Natürlich können die worst–case–Kosten für ($n\times$ *insert*)+($n\times$ *delete-min*) nicht unter $\Omega(n\log(n))$ liegen, denn diese Operationen zusammengenommen stellen einen Sortieralgorithmus dar, mit unterer Schranke $\Omega(n\log(n))$. Also ist die worst–case–Zeitkomplexität von *insert* und *delete-min* jeweils mindestens $\Omega(\log(n))$.

- Die Verwendung von Fibonacci–Heaps erlaubt eine Verbesserung der Komplexität von Algorithmen für minimale Spannbäume sowie von Dijkstra's kürzeste–Wege–Algorithmus, denn diese verwenden relativ häufig die *decrease-key*–Operation, welche durch Fibonacci heaps billig zu implementieren sind.
  In einem Algorithmus, bei dem die *delete*–und *delete-min*–Operationen nur einen geringen Anteil der Gesamtheit aller Operationen darstellen, sind Fibonacci heaps asymptotisch schneller als binomial queues.

Bemerkung: Priority Queues eignen sich grundsätzlich nicht für Suchoperationen, unabhängig von ihrer Implementierung. Es ist daher immer eine zusätzliche andere Datenstruktur, etwa ein Suchbaum, zu verwenden.

Bemerkung: Jedes Element besteht aus einem *Schlüssel (key)* und einem Wert. Der Wert (Beispiel: Telefonnummer) darf in einer priority queue <u>nur einmal</u> vorkommen. Es ist jedoch zulässig, daß derselbe Schlüssel (Beispiel: Name) mehreren verschiedenen Werten zugeordnet ist. Für die folgenden Betrachtungen sind gespeicherte Werte unter dieser Voraussetzung uninteressant, es werden nur die Schlüssel betrachtet.

Die Datenstruktur

Definition(Fibonacci heap):

- Ebenso wie binomial heaps sind Fibonacci–heaps Wälder, in deren Bäumen jeweils die Minimumsvariante der heap–Bedingung gilt, so daß die Wurzel das Element mit minimalem Schlüssel des Baums enthält.

- Für die Topologie der einzelnen Bäume gelten von vornherein keine expliziten Einschränkungen — ihr Aussehen ist völlig beliebig. Es wird sich jedoch herausstellen, daß sich "von selbst" ausschließlich Binomialbäume ergeben, wenn als Startsituation nur ein leerer oder überhaupt kein heap vorliegt und nur die unten definierten Operationen verwendet werden.

- Als *Rang* eines Knotens bezeichnet man seinen Grad, d.h. die Anzahl seiner Kinder.

- **Implementierung:**

  – Innerhalb jedes Baums besitzt jeder Knoten einen Zeiger auf seinen Vater (NIL im Falle der Wurzel) und einen Zeiger auf eines seiner Kinder (z.B. stets das erste). Alle Kinder sind in einer doppelt verketteten zirkulären Liste organisiert. Schließlich ist zu jedem Knoten sein Rang und ein Markierungsbit gespeichert.

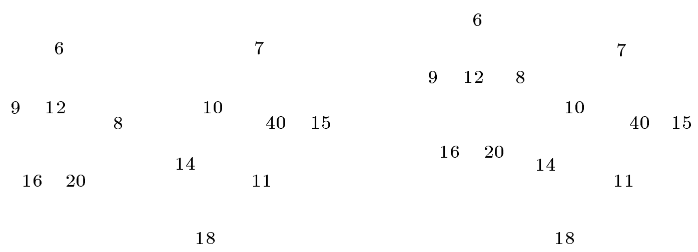
[8] werden später behandelt

– Die Wurzeln aller Bäume sind in einer zirkulären Liste doppelt verkettet. Auf die Wurzel mit minimalem Schlüssel[9] wird ein Zeiger, der *min–Pointer* gerichtet. Ist dieser Pointer NIL, so liegt ein leerer heap vor. Insgesamt benötigt jeder Knoten also 4 Pointer, einen Integer–Wert (Rang), ein Markierungsbit, sowie Platz für den Schlüssel und den Wert.

Bemerkung: Die doppelte Verkettung der Kinder– bzw. Wurzel–listen innerhalb von heaps[10] erlaubt das Löschen eines Listeneintrags in Zeit $O(1)$. Durch die zirkuläre Organisation wird andererseits ihre Konkatenation in konstanter Zeit ermöglicht.

Linken von Bäumen: Das Linken ist die grundlegende Operation von Bäumen mit Heap–Bedingung. Sie werden dabei so zusammengehängt, daß die Heap–Bedingung erhalten bleibt. Sind x und y die Wurzeln der zwei zu linkenden Bäume, so wird ein neuer Baum aufgebaut, dessen Wurzel bzgl. der Schlüssel das Minimum von x und y ist. Sei dies x. Dann erhält der Baum mit Wurzel x als zusätzlichen Unterbaum den Baum mit Wurzel y, d.h. der Rang von x erhöht sich um eins.

Beispiel:



Beobachtung: Die oben beschriebene Operation ist gerade diejenige, durch die der Binomialbaum B_1 aus zwei Bäumen B_0 und durch die B_k rekursiv aus zwei B_{k-1} entsteht. Deshalb entstehen bei geeigneter Startsituation ausschließlich Binomialbäume, wenn das Linken die einzige verwendete konstruktive Operation ist.

Die Operationen

Bemerkung: Im Vergleich zu binomial queues, bei denen sehr häufig gleichartige (Binomial)bäume verschmolzen werden, so daß nur B_is von verschiedenem Typ vorliegen, erfolgt dies bei Fibonacci heaps relativ selten, nämlich nur in der vergleichsweise teuren Operation *delete-min*, die dadurch gleichzeitig eine Aufräumfunktion erhält. Man sieht jedoch der Implementierung der anderen Operationen sofort an, daß sie die Datenstruktur recht unordentlich hinterlassen.

(i) Empty-Heap: Schaffen eines leeren Fibonacci heap: return NIL

$$\boxed{\text{Zeitkomplexität: } O(1)}$$

(ii) insert: Es wird ein neuer trivialer heap geschaffen, der nur aus einem Baum mit einer Wurzel (und diversen NIL–Pointern) besteht, und zum bestehenden heap gemerged. Das heißt, es wird in die Wurzelliste ein neuer Baum mit nur einem Knoten eingefügt. Eventuell muß der min–Pointer auf den neuen Knoten gerichtet werden.

---
[9] Dieses hat sogar von allen Elementen den minimalen Schlüssel

[10] nicht nur Fibonacci heaps

Zeitkomplexität: $O(1)$

(iii) merge: Diese Operation ist nur für disjunkte Fibonacci heaps definiert. Ihre Wurzellisten werden durch geeignete Anpassung der Pointer konkateniert, und ggf. wird der min–Pointer verbogen.

Zeitkomplexität: $O(1)$

(iv) find-min: Es wird das Element ausgegeben, auf das der min–Pointer zeigt. Dabei handelt es sich sicher um eine Wurzel.

Zeitkomplexität: $O(1)$

(v) delete-min: Wie bereits erwähnt, hat diese Operation auch Aufräumfunktion und ist daher recht kostspielig. Sei x der Knoten, auf den der min–Pointer zeigt.

```
Entferne x aus der Wurzelliste
 Konkateniere Liste der Kinder von x und Wurzelliste
 WHILE (es ex. mind. 2 Bäume mit gleichem Rang a.d. Wurzel) DO
   Linke die beiden Bäume
 OD
Suche neues Minimum unter den Wurzeln und passe min--Zeiger an
```

Man beachte, daß an jedem Knoten, insb. jeder Wurzel, der Rang gespeichert ist. Zwar vereinfacht dies die Implementierung, doch müssen noch immer Paare von Wurzeln gleichen Rangs gefunden werden. Dies wird wie folgt realisiert: Wir verwenden ein array, dessen Indizes je für einen Rang stehen. Die Elemente sind Zeiger auf entsprechende Wurzeln dieses Rangs. Es ist garantiert, daß ein Element nur dann unbesetzt ist, wenn tatsächlich keine Wurzel entsprechenden Rangs existiert. Nach dem Entfernen des Knotens x aus der Wurzelliste fügen wir die Kinder eines nach dem anderen in die Wurzelliste ein[11] und aktualisieren in jedem Schritt das array. Soll eine bereits besetzte Position des arrays beschrieben werden, so wird ein Link–Schritt ausgeführt und versucht, einen Pointer auf die neue Wurzel in die nächsthöhere Position im array zu schreiben. Dies zieht evtl. weitere Link–Schritte nach sich. Nach Abschluß dieser Operation enthält der Fibonacci heap nur verschiedene Bäume. Liegt der kanonische Fall vor, daß stets ausschließlich Binomialbäume im Spiel sind, so erhalten wir durch die *delete-min*–Operation eine binomial queue. Für die Kostenbetrachtung beachte man: Eine Situation mit maximaler Anzahl von Link–Schritten entspricht genau der des sukzessiven Einfügens einzelner Elemente in eine binomial queue. Dort wurde gezeigt, daß dies eine Komplexität von $O(N)$ hat. Insgesamt gilt:

Zeitkomplexität: $O(\text{max. Rang}) + O(\#\text{Link–Schritte})$

(vi) decrease-key: Die Operation *decrease-key*(Δ, i, h) bewirkt eine Verkleinerung des Schlüssels des Elements i um den Wert Δ innerhalb des heaps h. Die folgende Darstellung des 'Algorithmus' ist recht technisch und nur wenig tiefliegend. Er wird jedoch zunächst nur unvollständig angegeben und später erweitert. Sei der Knoten x mit dem zu manipulierenden Schlüssel nicht die Wurzel. Dann kann eine Verkleinerung des Schlüssels die Heap–Bedingung verletzen. Es wird nicht

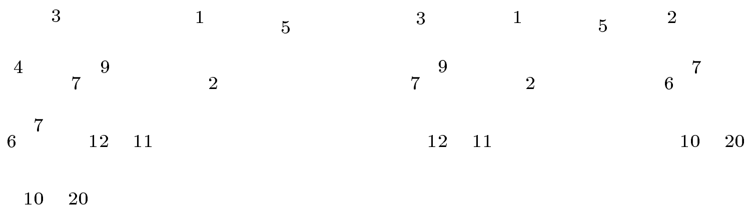
[11] Das elementweise Vorgehen wird durch das Suchen nach zwei Wurzeln mit gleichem Rang diktiert.

geprüft, ob dies eintritt, sondern im Sinne niedriger Zeitkomplexität der Knoten mit seinem Unterbaum aus seiner Position entfernt.

```
Verkleinere key(i) um Δ;
x:=Knoten mit Element i;
Lösche Zeiger von x zum Vater p(x) (:=NIL);
Lösche x in der Liste der Kinder von p(x);
füge x (samt Unterbaum) unmarkiert in Wurzelliste ein;
Verkleinere Rang(p(x)) um 1;
Verändere ggf. den min--Pointer;
```

Ist x eine Wurzel, so ist nur die erste Zeile auszuführen, da die Heap–Bedingung dann sicher erfüllt bleibt.

$$\boxed{\text{Zeitkomplexität: } O(1)}$$

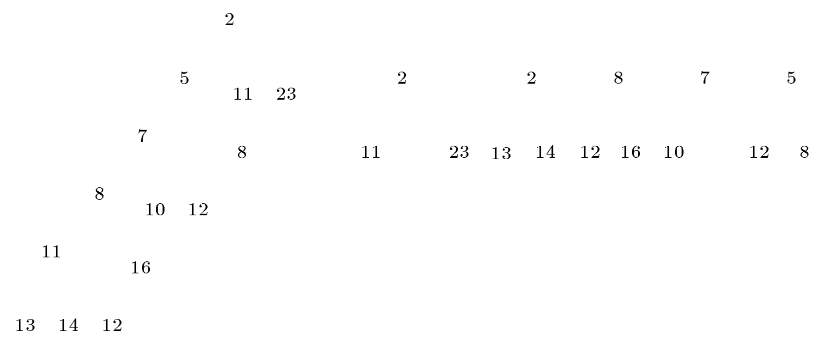


Bemerkung (kaskadiertes Abschneiden): Um die gewünschten Schranken für die amortisierten Kosten zu erreichen, vereinbaren wir: Wurde durch einen Link–Schritt (bei *delete-min*) eine Wurzel x zu einem Kind eines anderen Knotens gemacht, so gilt für x: Sobald x durch *decrease-key*–oder *delete*– Operationen zwei seiner Kinder verliert, so soll x von seinem Vater abgetrennt und in die Wurzelliste eingefügt werden, wie bei *decrease-key*.
Daher ist eine Markierung der Knoten nötig, die anzeigt, ob bereits ein Kind verloren wurde. In einem Link–Schritt bei *delete-min* wird an der ehemaligen Wurzel x, die neues Kind eines anderen Knoten wird, die Markierung gelöscht. Bei *decrease-key* und *delete* wird der Vater eines jeden abgeschnitteten Knotens markiert, da er ein Kind verliert. Ist er bereits markiert, so wird auch er (mitsamt seinen restlichen Kindern) abgeschnitten und in die Wurzelliste eingefügt. Somit verliert dessen Vater ein Kind und wird markiert. Eventuell war auch dieser bereits markiert, usw...

(vi) decrease-key `verkleinere key(i) um Δ;`

```
(((Falls y keine Wurzel ist:)))
x:=Knoten mit Element i;
y:=p(x);
entferne x;
füge x unmarkiert in Wurzelliste ein;
WHILE y markiert DO
  entferne y;
  füge y unmarkiert in Wurzelliste ein;
  IF p(y)=NIL THEN quit FI;
  y:=p(y);
OD;
markiere y;
```

Beispiel:



(vii) delete:

```
x:=Knoten mit Element i;
y:=p(x);
entferne x;
füge Liste der Kinder in Wurzelliste ein;
WHILE y markiert DO
  Entferne y;
  Füge y unmarkiert in Wurzelliste ein;
  IF p(y)=NIL THEN quit FI;
  y:=p(y);
OD;
markiere y;
```

Kosten für (vi) und (vii): Das eigentliche Löschen erfolgt in Zeit $O(1)$, das kaskadierte Abschneiden ist in jeder Ebene des betreffenden Baums höhstens einmal nötig. Daher:

$$\boxed{\text{Zeitkomplexität: } O(1) + O(\text{Höhe des Baums}) = O(\log(N))}$$

Zur Erinnerung: Wird mit einem leeren oder garkeinem Fibonacci heap gestartet und werden ausschließlich die aufbauenden Operationen (i) bis (v) angewendet, so können nur Binomialbäume entstehen. In diesem 'natürlichen' Fall also liegt stets ein Binomialwald vor, der jedoch i.a. nicht aufgeräumt ist. Das heißt, es existieren mehrere Binomialbäume B_i desselben Typs, die nicht paarweise zu B_{i+1}–Bäumen verschmolzen sind. Dies geschieht erst bei der Operation *delete-min*. Man beachte, daß nur nach einer solchen Operation momentan ein binomial heap vorliegt, ansonsten nur ein Binomialwald.

Amortisierte Kostenanalyse

Motivierung: Zur Abschätzung der Gesamtkosten einer Folge von Operationen erhält man durch die Summierung der worst–case–Kosten der einzelnen Operationen keine realistische Aussage; Der resultierende Wert ist zu groß. Deshalb werden die Einzelkosten auf einen Gesamtkontext bezogen betrachtet. Einige Operationen sind nämlich — für sich betrachtet — billig, zerstören jedoch die Organisation der Datenstruktur. Dazu gehört etwa die *insert*–Operation. Dagegen ist *delete-min* mit $O(N)$ teuer, stellt aber einen geordneten Zustand her. Diesem Umstand wird in der amortisierten Analyse Rechnung getragen.

Definition: Wir führen für jede Datenstruktur (Fibonacci heap) ein *Bankkonto* ein und ordnen ihr eine nichtnegative reelle Zahl *bal*, ihr *Potential* (bzw. *Kontostand*) zu.

Die *amortisierten Kosten* für eine Operation ergeben sich als Summe der tatsächlichen Kosten und der Veränderung $\Delta bal = bal_i - bal_{i-1}$ des Potentials, welche durch die Operation verursacht wird:

$$a_i = t_i + \Delta bal = t_i + bal_i - bal_{i-1}.$$

Folglich sind die amortisierten Kosten höher als die tatsächlichen Kosten, falls $\Delta bal > 0$. Das bedeutet, eine Operation (wie *insert*) wird teurer, wenn sie die Datenstruktur unordentlich hinterläßt. Und eine Operation, die die Datenstruktur ordentlicher hinterläßt, als sie sie vorgefunden hat (deshalb $\Delta < 0$) ist amortisiert billiger. Für Operationen, die den heap nicht verunstalten, aber auch nicht aufräumen, sind tatsächliche und amortisierten Kosten gleich. Es ergibt sich, daß für eine Folge von Operationen die gesamten amortisierten Kosten als Summe aller tatsächlichen Kosten und der gesamten Veränderung des Potentials zu erhalten sind. Deshalb gilt:

Lemma: Falls — wie in der Definition gefordert — bal niemals negativ wird und mit dem Wert $bal_0 = 0$ startet, so ist bei einer Folge von Operationen die Summe der amortisierten Kosten eine obere Schranke für die gesamten tatsächlichen Kosten.

Beweis: Es gilt:

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (t_i + bal_i - bal_{i-1}) = \sum_{i=1}^{m} t_i + bal_m - bal_0 \geq \sum_{i=1}^{m} t_i,$$

denn $bal_m \geq 0$ und $bal_0 = 0$, *q.e.d.*

Definieren wir zunächst das Potential bal eines Fibonacci heaps als die Anzahl seiner Bäume, also die Länge der Wurzelliste. Dann ergibt sich:

Operation	tatsächlich	amortisiert
insert	$O(1)$	$O(1)$
find-min	$O(1)$	$O(1)$
merge	$O(1)$	$O(1)$

Eine *insert*–Operation erhöht bal um 1, also eine Konstante; Die anderen Operationen verändern nichts am Potential der Datenstruktur.
Nun wird die Definition des Potentials modifiziert, so daß sie auch die Markierung der Knoten und somit ihre Anfälligkeit für kaskadierte Schnitte berücksichtigt:

$$bal := \#\ \text{Bäume} + 2 \cdot \#\ (\text{markierte Knoten} \neq \text{Wurzel})$$

Man sieht, daß die *einzelnen* amortisierten Kosten eine niedrigere Abschätzung als bei der worst–case–Betrachtung darstellen:

Operation	tatsächlich	amortisiert
insert	$O(1)$	$O(1)$
find-min	$O(1)$	$O(1)$
merge	$O(1)$	$O(1)$
decrease-key	$O(\log(N))$	$O(1)$
delete	$O(\log(N))$	$O(\log(N))$
delete-min	$O(N)$	$O(\log(N))$

Beweis: Für die ersten drei Operationen bleiben die amortisierten Kosten von der neuen Definition unbeeinflußt, denn es ändert sich dabei nichts an der Markierung der Knoten. Für die anderen Operationen:

Lemma: Sei x ein Knoten im Fibonacci heap, und seien die Kinder von x in der Reihenfolge ihres Einfügens (bei Link–Schritten) angeordnet. Dann hat das i-te Kind von x mindestens $(i-2)$ Kinder, d.h. $Rang = (i-2)$.

Beweis: Zum Zeitpunkt des Einfügens des i-ten Kindes y an x hat x mindestens $(i-1)$ Kinder (einige davon sind nach dem Link–Schritt vielleicht verloren gegangen). Ferner ist bekannt, daß x und y beim Link–Schritt den gleichen Rang hatten, denn das veranlaßte ja das Zusammenfassen. Also hatte auch y mindestens den Rang $(i-1)$, jedenfalls unmittelbar nach dem Linken. Danach jedoch konnte sich $Rang(y)$ noch um bis zu 1 verkleinern, ohne dadurch ein kaskadiertes Abschneiden von y auszulösen. Folglich: $Rang(y) \geq (i-2)$, *q.e.d.*

Corollar: Ein Knoten x mit Rang k hat mindestens $fib(k+2) \geq \Phi^k$ Nachfahren, sich selbst eingeschlossen. Dabei gilt $fib(0) = 0$, $fib(1) = 1$, $fib(k) = fib(k-2) + fib(k-1)$ und $\Phi = \frac{1+\sqrt{5}}{2}$.

Beweis: Sei S_k die kleinstmögliche Anzahl von Nachkommen von x, sich selbst eingeschlossen, wenn $Rang(x) = k$. Dann gilt:
•$S_0 = 1$ und $S_1 = 2$ (offensichtlich);
•$S_k \geq \sum_{i=0}^{k-2} S_i + 2$ für $k \geq 2$ (nach Lemma);
Es ist andererseits $fib(k+2) = \sum_{i=2}^{k} fib(i) + 2$ für $k \geq 2$. Durch Induktion über k erhält man $S_k \geq fib(k+2)$ für alle $k \geq 0$. Ferner sieht man durch Entwicklung der Rekursionsgleichung für fib, daß $fib(k+2) \geq \Phi^k$, *q.e.d.*

Forts. Beweis: Mit Hilfe der obigen Aussagen erhält man:

delete-min: Sei $k := Rang(x)$, dann hat x mindestens Φ^k Nachfahren, und es gilt: $\Phi^k \leq N$. Folglich:

$$k \leq \log_\Phi(N) = \frac{\log(N)}{Log(\Phi)} \leq 1.4404 \cdot \log(N).$$

Durch das Einfügen aller Kinder in die Wurzelliste erhöht sich *bal* um $\leq 1.4404 \cdot \log(N)$ und verringert sich durch das Linken um die Zahl der Link–Schritte. Die Kosten für die Verringerung werden vom Bankkonto bezahlt, machen sich also amortisiert nicht bemerkbar. Die amortisierten Kosten für *delete-min* betragen also $O(\log(N))$.

decrease-key: Es erhöht sich *bal* um $(3 - \#$ kaskadierte Schnitte$)$, denn beim ersten Schnitt wird ein (eventuell unmarkierter) Knoten zu einer Wurzel, was $bal := bal + 1$ bewirkt $(*)$. Bei jedem kaskadierten Schnitt[12] wird ein Knoten zu einer Wurzel ($\Longrightarrow bal := bal + 1$). Er fällt dadurch als markierter Knoten fort, was im Potential mit Faktor 2 berücksichtigt wird. Also: $bal := bal - 2$, und insgesamt: $bal := bal - 1$ $(**)$. Im letzten Schritt wird ein unmarkierter Knoten markiert und folglich $bal := bal + 2$ $(***)$. Man erhält insgesamt:

$$\Delta bal = (*) + (\#\text{ kask. Schnitte} \cdot (**)) + (***) = 3 - \#\text{ kask. Schnitte}$$

Und die amortisierten Kosten betragen $O(1)$.

delete: Man kombiniere *decrease-key* und *delete-min* und erhält so die amortisierten Kosten von *delete* als $O(\log(N))$.

[12]Eventuell gibt es garkeine, und der erste Schnitt ist zugleich der letzte

Zusammenfassung: Die gesamten tatsächlichen Kosten für eine Folge von Operationen sind höchstens so groß wie die Summe ihrer amortisierten Kosten, nämlich $O(\log(N))$ für *delete-min* und *delete*, und $O(1)$ für *insert, merge, find-min* und *decrease-key.*

3.7 Selbstorganisierende lineare Listen

Beim sog. "Wörterbuchproblem" ist auf möglichst effiziente Weise eine Menge von bis zu n Elementen unter den Operationen

access(i): Suchen des Elements i

insert(i): Einfügen des Elements i

delete(i): Löschen des Elements i

zu organisieren. Dabei soll die Effizienz der Implementierung bei beliebigen Sequenzen von m Operationen untersucht werden. Die Menge wird als unsortierte Liste dargestellt, und die Operationen werden wie folgt implementiert:

access: Die Liste wird von Anfang an durchsucht, bis das gesuchte Element gefunden ist. Für das i–te Element betragen die tatsächlichen Kosten i Einheiten.

insert: Die Liste wird von Anfang an durchsucht, und wenn sie vollständig durchsucht wurde und das Element nicht enthalten ist, so wird es hinten angefügt. Dies erfordert $(i+1)$ Schritte, wenn die Länge der Liste i beträgt.

delete: Wie bei *access* wird die Liste zuerst durchsucht und dann das betreffende Element gelöscht. Dies kostet im Falle des i–ten Elements i Schritte.

Nach Abschluß einer jeden Operation können gewisse Umordnungsschritte stattfinden, die spätere Operationen beschleunigen.

Annahme: Wir nehmen an, daß nach Ausführung jeder *access*– oder *insert*–Operation, die auf Element i angewendet wird, dieses Element kostenlos um beliebig viele Positionen in Richtung Listenanfang geschoben werden kann[13].Wir nennen jede daran beteiligte Elementenvertauschung einen *kostenlosen Tausch.* Jeder andere Tausch zweier Elemente ist ein *bezahlter Tausch* und kostet 1 Zeiteinheit.

Ziel: Es soll eine einfache Regel gefunden werden, durch die mittels obiger Vertauschungen die Liste so organisiert wird, daß die Gesamtkosten einer Folge von Operationen minimiert werden. Wir sprechen von "*selbstorganisierenden linearen Listen*" und untersuchen hierzu folgende Regeln:

(i) MTF (Move–to–Front): *Access* und *insert* stellen das betreffende Element vorne an die Liste an und verändern ansonsten nichts.

(ii) TR (Transpositionsregel): *Access* bringt das Element durch eine Vertauschung um eine Position weiter nach vorne, *insert* hängt es ans Ende der Liste an.

Behauptung: Die MTF–Regel erlaubt die Ausführung einer Sequenz von Operationen *access*, *insert* und *delete* mit einer amortisierten Laufzeit, die um höchstens den konstanten Faktor 2 schlechter ist als ein optimaler die Sequenz implementierender Algorithmus. Für die TR dagegen gibt es keinen solchen konstanten Faktor; Sie kann verglichen mit MTF bzw. einem optimalen Algorithmus beliebig schlecht sein.

[13]Für oft benötigte Elemente ist eine Position nahe am Listenanfang günstig.

Schreibweisen: Sei s eine beliebige Folge von Operationen des Typs *access*, *insert* und *delete*. Diese Folge soll auf die anfangs leere Liste angewendet werden. Mit A bezeichnen wir einen beliebigen optimalen Algorithmus, der die betrachtete Folge von Operationen implementiert. Die im folgenden bewiesenen Aussagen gelten zwar für jeden beliebigen Algorithmus A, aber daher speziell auch für einen optimalen, was die Herleitung des Faktors 2 in obiger Aussage erlaubt. Die gesamten tatsächlichen Kosten von A, ohne Berücksichtigung bezahlter Tauschvorgänge[14], werden mit $C_A(s)$ dargestellt, ferner sei $X_A(s)$ die Anzahl der bezahlten Vertauschungen für die Folge s im Algorithmus A und $F_A(s)$ die Anzahl der unbezahlten Vertauschungen. Für den MTF–Algorithmus zu s verwenden wir entsprechend die Variablen $C_{MTF}(s)$, $X_{MTF}(s)$ und $F_{MTF}(s)$. Schließlich sei m die Länge der Folge s, also die Anzahl der Operationen.

Bemerkung: Nach Definition der Regeln gilt:

- $X_{MTF}(s) = 0$
- $X_{TR}(s) = 0$
- $F_A(s) \leq C_A(s) - m$

Beweis: Für die dritte Aussage bedenke man, daß nach *access* bzw. *insert*, angewendet auf das i-te Element, bis zu $(i-1)$ kostenlose Vertauschungen möglich sind. Somit ist F_A etwa dann maximal, wenn m-mal *access* angewendet wird, jeweils auf das letzte Listenelement, und wenn jeweils $(n-1)$ kostenlose Vertauschungen folgen. Dann ist $F_A = m \cdot (n-1) = m \cdot n - m$. Der Aufwand $C_A = m \cdot n$, da jede Anwendung von *access* n Einheiten kostet. So ergibt sich im maximalen Fall $F_A = C_A - m$, *q.e.d.*

Satz: Für jede Sequenz s der Länge m von Operationen, beginnend mit der leeren Liste, gilt:

$$C_{MTF}(s) \leq 2 \cdot C_A(s) + X_A(s) - F_A(s) - m.$$

Das bedeutet, MTF ist höchstens doppelt so schlecht wir jeder optimale Algorithmus.

Beweis:

Bezeichnungen: Wir verwenden wieder das Bankkonten–Paradigma und kürzen die tatsächlichen Kosten für die Operation i mit t_i ab, ihre amortisierten Kosten durch a_i. Zur Definition des Potentials wählen wir ein neues Vorgehen: Es mögen der MTF–Algorithmus und ein beliebiger (optimaler) Algorithmus parallel laufen, startend mit der leeren Liste. Wir betrachten die Situation nach Ablauf eines Präfixes s' der Folge s, welche bei MTF die Liste L_{MTF} und bei A die Liste L_A erzeugt. Dann betrachten wir das *Potential von* L_{MTF} *bzgl. der Liste* L_A:

$$bal(L_{MTF}, L_A) := \# \text{ der inversen Stellen in } L_{MTF} \text{ bzgl } L_A$$

Eine *inverse Stelle* ist ein Vorkommen zweier Elemente, die in den beiden Folgen in verschiedener Reihenfolge vorkommen. Sie müssen in keiner der Folgen benachbart sein.

Lemma: Bezüglich dieser Potentialfunktion verursachen die Operationen bei MTF folgende amortisierten Kosten:

access auf das Element auf Position i in L_A:	$\leq 2i - 1$ Einheiten
insert in eine Liste mit i Elementen:	$\leq 2(i+1) - 1$ Einheiten
delete des Elements auf Position i in L_A:	$\leq i$ Einheiten
Eine unbezahlte Vertauschung in L_A:	-1 Einheit
Eine bezahlte Vertauschung in L_A:	≤ 1 Einheit

[14]Die unbezahlten Vertauschungen gehen ohnehin in keine Rechnung ein.

Die Startkonfiguration hat das Potential 0 (leere Listen), eine Endkonfiguration hat stets ein nichtnegatives Potential. Wie üblich impliziert dies, daß die tatsächlichen Kosten einer Folge von Operationen beschränkt sind durch die Summe der amortisierten Kosten.

Lemma: Für eine *access*– oder *insert*–Operation ergeben sich bei einem MTF–Algorithmus die amortisierten Kosten $(2c_A - 1)$, wenn c_A die Kosten der Operation im Algorithmus A sind. Der amortisierte Aufwand für *delete* ist $\leq 2c_A - 1$. Die m Summanden (-1) ergeben zusammen $(-m)$.

Beweis: Sei i die Position des Elements x_i in L_A und t in L_{MTF}. Dann kostet eine Operation bei A höchstens i Zeiteinheiten und höchstens t bei MTF. Sei k die Anzahl der Elemente, die in L_{MTF} vor x_i stehen, in L_A jedoch hinter x_i. Dann folgt $t - k \leq i \Longleftrightarrow k \geq t - i$. Weiter folgt $\Delta bal = -k \leq i - t$. Wegen $a = t + \Delta bal$ gilt für die amortisierten Kosten a: $a \leq 2i - 1 = 2c_A - 1$.

Hieraus folgt die Ungleichung des Satzes. Denn die m Summanden (-1) ergeben zusammen $(-m)$, und es sind noch $X_A(s) \times (+1)$ und $F_A(s) \times (-1)$ zu addieren, denn bezahlte bzw. unbezahlte Vertauschungen bewirken die Potentialveränderung $\Delta bal = +1$ bzw. -1. Insgesamt ergibt sich also

$$C_{MTF}(s) \leq A_{MTF}(s) \leq 2 \cdot C_A(s) - m + X_A(s) - F_A(s), q.e.d.$$

Corollar: Wird mit einer nichtleeren Liste begonnen, so gilt:

$$C_{MTF}(s) \leq 2 \cdot C_A(s) - m + X_A(s) - F_A(s) + \underbrace{bal(L_{MTF}, L_A)}_{\leq n^2}.$$

Das bedeutet insbesondere, daß ein MFT–Algorithmus $O(n^2)$ Schritte benötigt, um im worst case die unvorteilafte Anfangskonfiguration auszugleichen. Danach sind die Kosten höchstens doppelt so hoch wie bei einem optimalen Algorithmus für s.

3.8 Selbstorganisierende binäre Suchbäume — Splay Trees

3.8.1 Vorbemerkungen

Wie schon in früheren Kapiteln sollen binäre Suchbäume verwendet werden, um Elemente, bestehend aus einem Schlüssel und einer Information, anhand ihres Schlüssels zu lokalisieren. Das Universum ist total geordnet, und innerhalb eines Suchbaums soll die Invariante "Knoten im $\mathbb{lUb}$ < key(Wurzel) < Knoten im $\mathbb{rUb}$" gelten. Damit ist ein *divide & conquer*–Algorithmus anwendbar, der einen Knoten in Tiefe d mit dem Zeitaufwand $\Theta(d)$ findet. Die worst–case–Abschätzung ist natürlich $O(\log(n))$ bis $O(n)$ für einen n-elementigen nicht notwendig balancierten Suchbaum. Doch selbst die bisher behandelten Arten von balancierten Bäumen, wie die höhenbalancierten Suchbäume oder B–Bäume, haben gewisse Nachteile: Für die Speicherung der Balancierungsinformation wird extra Platz benötigt, und generell sind diese Datenstrukturen darauf ausgelegt, die worst–case–Kosten pro Operation zu minimieren (auf $O(\log(n))$ im Falle balancierter binärer Suchbäume), während in der Praxis eher Sequenzen von Operationen vorliegen. Also erscheint es günstiger, die gesamte Laufzeit solcher Sequenzen zu minimieren. Daher betrachten wir die amortisierten Kosten, welche ja für eine Operation deren durchschnittliche Kosten im Kontext einer worst–case–Sequenz von Operationen angeben.

Der hier zugrundeliegende Ansatz bemüht sich, innerhalb einer Folge von Operationen die einzelnen Operationen durch Realzeit–Umorganisation der Datenstruktur möglichst weit vom worst case zu entfernen, statt die jeweiligen worst–case–Kosten zu minimieren. Bei jedem Zugriff auf den Baum soll eine möglichst einfache Reorganisationsregel angewendet werden, um die Laufzeit künftiger Operationen zu vermindern. Wie bei der MTF–Regel für selbstorganisierende Listen soll erreicht werden, daß Elemente mit häufigem Zugriff möglichst nahe an der Wurzel liegen. Ein Element, auf das die Suchoperation angewendet wird, wird zur Wurzel befördert — in der i.a. berechtigten Hoffnung, daß weitere Zugriffe auf dieses Element folgen.

Definition: Ein *splay tree* ist ein interner binärer Suchbaum, d.h.

(i) Knoten im $\mathbb{lUb}$ < key(Wurzel) < Knoten im $\mathbb{rUb}$

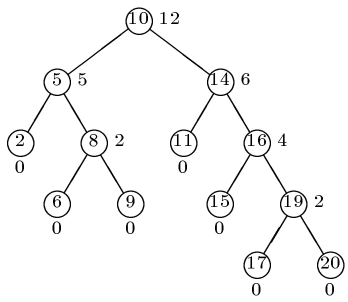
(ii) Alle Knoten tragen Elemente, denn Zeiger auf Blätter sind NIL–Pointer.

Die zur Datenstruktur gehörigen Operationen sind *access, splay, insert, delete, join2, join3, split, change-weight* und werden später definiert.

Die *in–order–Position* eines Elements ist die um 1 erhöhte Anzahl der Elemente, die ihm bei in–order–Linearisierung vorausgehen. Früher wurde gezeigt, daß diese Linearisierung eine echt[15] aufsteigende Folge produziert.

Anmerkung: Durch eine kleine Erweiterung der Datenstruktur können Elemente auch anhand ihrer in–order–Position lokalisiert werden. Dazu speichert man an jedem Knoten zusätzlich die Anzahl seiner Nachkommen ab.

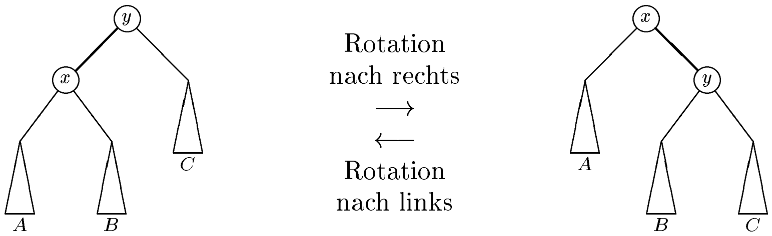
[15] Hier gilt auf Grund der Forderung (i) eine strikte Ungleichheit



Folgender Algorithmus implementiert die Suche eines Elements mit Tiefe d in Zeit $\Theta(d)$, was je nach Topologie des Baums $O(\log(n))$ bis $O(n)$ bedeutet. Dabei sei k seine in–order–Position.

```
v:=Wurzel;
if k<=#(v) then
 while (v!=Blatt) do
  if k=(#(v.lUb)+2) then exit
  elsif k<=(#(v.lUb)+1) then v:=v.lUb
  else k:=k-#(v.lUb)-2; v:=v.rUb
 od;
 return v;
fi;
```

Rotation: Die im nächsten Abschnitt eingeführte Operation *splay* dient dazu, ein Element, sobald es einen Zugriff erfährt, an die Wurzel des Suchbaums zu schaffen und auch für alle anderen Elemente auf dem Pfad zur Wurzel deren Tiefe zu reduzieren. Diese Operation wird mit Hilfe von Rotationen implementiert, und eine Rotation ist wie folgt definiert:



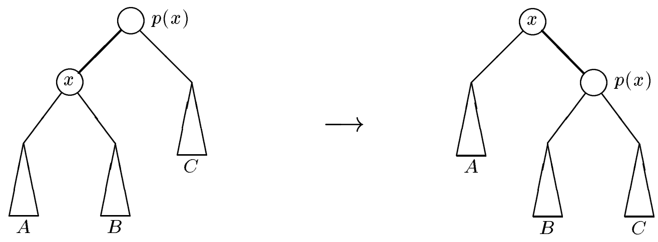
(Rotation um die Kante zwischen x und y)

Die Rotation bewahrt die symmetrische Ordung, d.h. die Sortiertheitsinvariante des Suchbaums. Sie kommt mit dem Zeitaufwand $O(1)$ aus. Für die folgende Rechnung nehmen wir an, sie koste 1 Einheit.

3.8.2 Strukturanpassung durch *splaying*

Die Operation, durch die einem Element, welches einen Zugriff erfährt, im Baum ein günstigerer Platz zugewiesen wird, heißt *splaying* und wird durch Rotationen realisiert, die in den meisten Fällen paarweise und auf dem Weg vom betreffenden Element zur Wurzel ausgeführt werden. Es gibt drei Fälle:

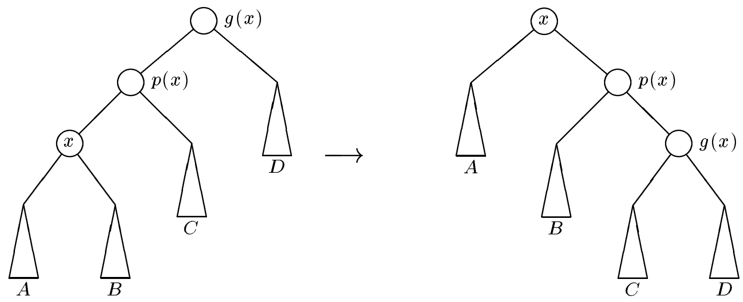
1. Fall: Der Vater $p(x)$ des Elements x ist die Wurzel des Baums (Zick–Fall, terminal); Es wird eine einfache Rotation der Kante zwischen x und $p(x)$ ausgeführt.



2. Fall: $p(x)$ ist nicht die Wurzel des Baums; x und $p(x)$ sind beides rechtes oder beides linkes Kind von $p(x)$ bzw. Großvater $g(x)$ (Zick–Zick–Fall). Durchgeführt werden

(i) Rotation der Kante zwischen $p(x)$ und $g(x)$

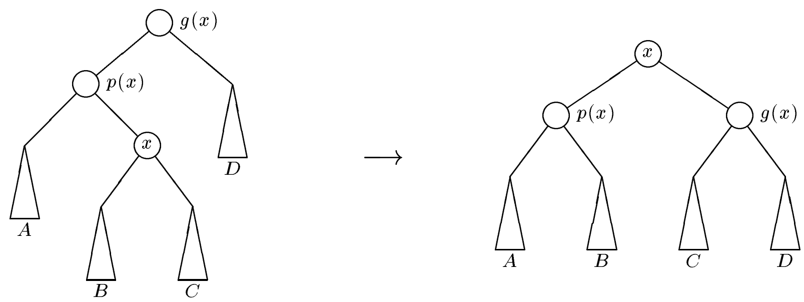
(ii) Rotation der Kante zwischen x und $p(x)$



3. Fall: $p(x)$ ist nicht die Wurzel, und x bzw. $p(x)$ sind rechtes bzw. linkes Kind von $p(x)$ bzw. $g(x)$, oder umgekehrt. (Zick–Zack–Fall) Dann ist folgendes zu tun:

(i) Rotation der Kante zwischen x und $p(x)$

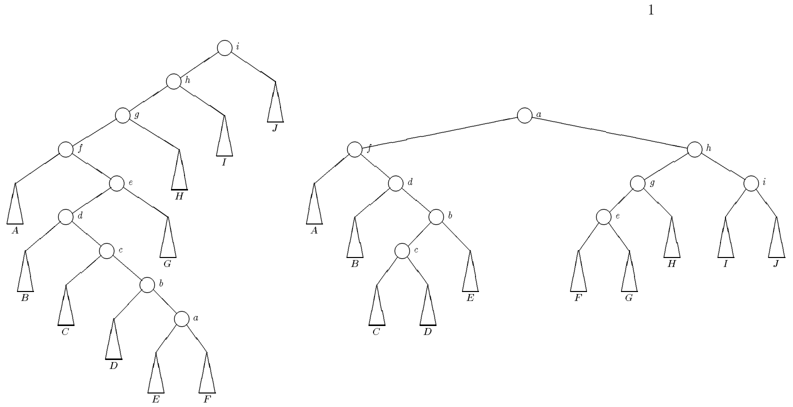
(ii) Rotation der Kante zwischen x und $g(x)$



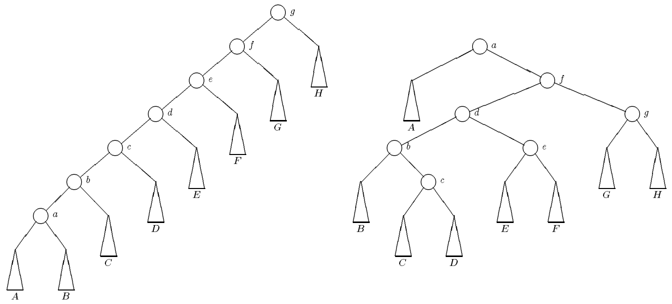
Dieses Vorgehen wird auf x von seiner ursprünglichen Position an rekursiv solange wiederholt, bis der terminale Fall ausgeführt und x Wurzel ist.

Bemerkung: Die Operation *splay(x)* befödert den Knoten x an die Wurzel des binären Suchbaums und bewirkt außerdem, daß die Tiefe aller Elemente, die auf dem Pfad liegen, etwa halbiert wird. Die Kosten von *splay(x)* betragen $\Theta(d)$, wenn sich x in Tiefe d befindet. Je nach Topologie des Baums liegt dies zwischen $O(\log(n))$ und $O(n)$. Der Aufwand ist also proportional zum Aufwand für *access(x)*.

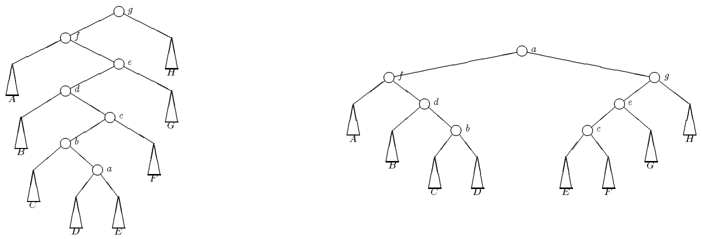
Beispiel:



Wirkung einer *splay*–Operation am Knoten a



Extremsituation für Fall 2: *splay(a)*



Extremsituation für Fall 3: *splay(a)*

3.8.3 Amortisierte Kosten der Operation *splay*

Zur Bestimmung der amortisierten Kosten von *splay* wird wie üblich eine Potentialfunktion definiert, die jeder Konfiguration der Datenstruktur eine reelle Zahl zugeordnet, ihr Potential bal, und wir definieren die amortisierten Kosten a_i, abhängig von ihren tatsächlichen Kosten t_i, als $a_i = t_i + bal_i - bal_{i-1} = t_i + \Delta bal$. Es folgt: $\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i + bal_0 - bal_m$. Ist $bal_m \geq bal_0$, so sind die gesamten amortierten Kosten eine obere Schranke für die Summe der tatsächlichen Kosten.

Sei jeder Schlüssel i mit einem beliebigen aber festen Gewicht $w(i)$ versehen. Für einen Knoten x sei die "Größe" $s(x)$ die Summe aller Gewichte im Unterbaum mit Wurzel x, und sein "Rang" $r(x)$ ist definiert durch $r(x) = \log(s(x))$. Wir definieren das Potential *bal* eines splay tree als die Summe der $r(x)$ aller Knoten x, d.h.:

$$bal = \sum_{\text{Knoten } x} \log\left(\sum_{y\in \mathrm{UB}(x)} w(y)\right)$$

Satz 0: Die amortisierten Kosten für *splay(x)* in einem Baum mit Wurzel t sind höchstens

$$3\,(r(t)-r(x))+1 = O\left(1+\log\left(\frac{s(t)}{s(x)}\right)\right).$$

Beweis: Falls x bereits die Wurzel ist, so gilt diese Aussage offensichtlich. Also möge mindestens eine Rotation stattfinden. Für einen *splay*–Schritt (mit 1 oder 2 Rotationen) seien s und r bzw. s' und r' die Funktionen für Größe und Rang von Knoten vor bzw. nach diesem Schritt.
Beh.: In Fall 1 betragen die amortisierten Kosten $\leq 3(r'(x)-r(x))+1$, und in Fall 2 und 3 ist $3(r'(x)-r(x))$ eine obere Schranke.
Bew.: Sei $y := p(x)$ und in Fall 2 und 3 $z := g(x) = p(p(x))$.

1. Fall: Es wird eine Rotation ausgeführt ($\Rightarrow$ Summand $+1$); Nur x und y ändern ihren Rang, was den Summanden $1+r'(x)-r(x)+r'(y)-r(y)$ erzeugt. Für die Summe gilt:

$$\begin{aligned} 1+r'(x)-r(x)+r'(y)-r(y) &\leq 1+r'(x)-r(x) \text{ wegen } r(y)\geq r'(y)\\ &\leq 1+3(r'(x)-r(x)) \text{ wegen } r'(x)\geq r(x)\end{aligned}$$

2. Fall: Die zwei Rotationen sorgen für den Summanden $+2$, und es ändern x, y und z ihren Rang, wodurch der Summand $r'(x)-r(x)+r'(y)-r(y)+r'(z)-r(z)$ entsteht. Es gilt:

$$\begin{aligned} 2+r'(x)-r(x)+r'(y)-r(y)+r'(z)-r(z) &= 2+r'(y)+r'(z)-r(x)-r(y)\\ &\quad\text{wegen } r'(x)=r(z)\\ &\leq 2+r'(x)+r'(z)-2\cdot r(x)\\ &\quad\text{wegen } r(y)\geq r(x),\ r'(x)\geq r'(y)\\ &\leq 3(r'(x)-r(x))\end{aligned}$$

Die letzte Ungleichung verlangt $2r'(x)-r(x)-r'(z)\geq 2$ $(*)$. Es ist nach Definition $r(x)=\log(s(x))$, und die auf das Dreieck $x,y>0;\ x+y\leq 1$ eingeschänkte Funktion $f(x,y)=\log(x)+\log(y)$ besitzt das Maximum (-2) im Punkt $(\frac{1}{2},\frac{1}{2})$. Folglich ist

$$r(x)+r'(z)-2r'(x) = (r(x)-r'(x))+(r'(z)-r'(x)) = \log\left(\frac{s(x)}{s'(x)}\right)+\log\left(\frac{s'(z)}{s'(x)}\right) \leq -2,$$

denn $\frac{s(x)}{s'(x)}+\frac{s'(z)}{s'(x)}\leq 1$ wegen $s(x)+s'(z)\leq s'(x)$. Damit ist $(*)$ bewiesen.

3. Fall: Wir erhalten als amortisierte Kosten

$$\begin{aligned} 2+r'(x)-r(x)+r'(y)-r(y)+r'(z)-r(z) &\leq 2+r'(y)+r'(z)-2\cdot r(x)\\ &\quad\text{wg. } r'(x)=r(z),\ r(x)\leq r(y)\\ &\leq 2(r'(x)-r(x))\end{aligned}$$

Dies folgt aus $2r'(x)-r'(y)-r'(z)\geq 2$, was wie in Fall 2 mit Hilfe von $s'(y)+s'(z)\leq s'(x)$ und der Funktion f hergeleitet werden kann. Als amortisierte Kosten erhalten wir schließlich eine obere Schranke von $2(r'(x)-r(x))\leq 3(r'(x)-r(x))$.

Der Satz folgt durch Aufsummieren der amortisierten Kosten für alle *splay*–Schritte. Da Fall 1 nur einmal und als terminaler Fall vorkommt, ergibt sich eine echte Teleskopsumme mit Wert $\leq 3(r'(x) - r(x)) + 1 = 3(r(t) - r(x)) + 1$, wobei r bzw. r' die Funktionen für den Rang vor bzw. nach der *splay*–Operation sind, *q.e.d.*

3.8.4 Tatsächliche Laufzeit für eine Folge von *access*–Operationen

Im folgenden führen wir mehrere unabhängige Analysen der tatsächlichen Laufzeit für eine Folge von m *access*–Operationen im splay tree durch. Wir erhalten mehrere obere Schranken, die einen Vergleich mit statischen balancierten binären Suchbäumen und anderen Datenstrukturen zulassen.[16]
Wir nutzen hierfür aus, daß die Gewichte $w(i)$ nur Parameter in der Kostenanalyse, für die Datenstruktur als solche jedoch nicht bedeutsam sind. Obiger Satz gilt für beliebige Zuweisungen positiver Gewichte an die Schlüssel des splay tree. Wir wählen daher für jede der folgenden Aussagen eine günstige Gewichtsfunktion w.

Satz 1: Eine Folge von m *access*–Operationen hat als gesamte Laufzeit

$$O((m+n) \cdot \log(n) + m).$$

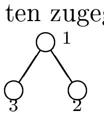
Beweis:

Lemma: Bleiben die Gewichte aller Knoten constant, so bewirkt eine Sequenz von m *access*–Operationen eine *Verkleinerung* des Potentials um höchstens den Wert

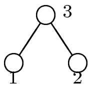
$$\sum_{i=1}^{n} \log(\frac{W}{w(i)}),$$

wobei wir definieren: $W = \sum_{i=1}^{n} w(i)$.

Anmerkung: Das Potential nimmt grundsätzlich dann ab, wenn auf schwere Knoten zugegriffen wird, so daß diese im Baum eine größere Höhe erhalten.



$bal = \log(3) + \log(2) + \log(6)$



$bal = \log(1) + \log(2) + \log(6)$

Beweis(Lemma): Es ist nach Definition $bal = \sum_{j=1}^{n} \log(s(i_j))$. Da für jeden Schlüssel i_j gilt: $w(i_j) \leq s(i_j) \leq (W)$, folgt für die maximale Potentialveränderung:

$$\Delta_{max} = \sum_{j=1}^{n} \log(W) - \sum_{j=1}^{n} \log(w(i_j)) = \sum_{j=1}^{n} \log\left(\frac{W}{w(i_j)}\right).$$

Forts.(Beweis): Wir setzen $w(i_j) := \frac{1}{n}$ für alle j. Dann ist $W = 1$, die amortisierte Zeit für einen *access* auf ein beliebiges Element ist $\leq (3\log(n) + 1)$, und die Potentialverkleinerung ist nach obigem Lemma maximal $n\log(n)$. Weil für die tatsächlichen Kosten im Vergleich zu den amortisierten Kosten gilt: $t = a - \Delta$, folgt: $t \leq 3m\log(n) + m + n\log(n) \leq 3(m+n)\log(n) + m = O((m+n)\log(n) + m)$, *q.e.d.*

[16] Für eine vollständige Behandlung siehe *Tarjan, Sleator: "Self–Adjusting Binary Search Trees"*, ACM Journal, Vol. 32, No. 3, pp. 659-660

Satz 2: Wird auf jeden Schlüssel mindestens $1\times$ zugegriffen, so ist die tatsächliche Laufzeit für m *access*–Operationen

$$O\left(m + \sum_{i=1}^{n} q(i) \cdot \log\left(\frac{m}{q(i)}\right)\right),$$

wobei $q(i)$ die Anzahl der Zugriffe auf Schlüssel i ist.

Beweis: Setzen wir $w(i) := \frac{q(i)}{m}$. Dann ist $W = 1$, die amortisierte Zugriffszeit für Schlüssel i ist $\leq 3 \cdot \log\left(\frac{m}{q(i)}\right) + 1$, und die Potentialsveränderung ist $\Delta bal \geq -\sum_{i=1}^{n} \log(\left(\frac{m}{q(i)}\right))$ nach dem Lemma im Beweis zu Satz 1. Folglich gilt:

$$t \leq \sum_{i=1}^{n} q(i) \cdot 3\log\left(\frac{m}{q(i)}\right) + m + \sum_{i=1}^{n} \log\left(\frac{m}{q(i)}\right) = \sum_{i=1}^{n} (3 \cdot q(i) + 1) \cdot \log\left(\frac{m}{q(i)}\right) + m$$

Dies ist $O\left(m + \sum_{i=1}^{n} q(i) \cdot \log\left(\frac{m}{q(i)}\right)\right)$, *q.e.d.*

Satz 3: Wir numerieren die Schlüssel im Baum in in–order–Reihenfolge von 1 bis n durch. Sei f ein fest gewählter Schlüssel und $i_1, \cdots, i_m$ die Folge der Schlüssel, auf die durch *access* zugegriffen wird. Dann ist die tatsächliche Zugriffszeit

$$O\left(n \cdot \log(n) + m + \sum_{j=1}^{m} \log(|i_j - f| + 1)\right).$$

Beweis: Nun setzen wir $w(i) := \frac{1}{(|i-f|+1)^2}$, so daß gilt: $W = \sum_{i=1}^{n} w(i) \leq \sum_{k=1}^{n} \frac{1}{k^2} = O(1)$. Ferner erhält man als amortisierte Kosten für einen Zugriff durch *access(i_j)* die Abschätzung $O(log(|i_j - f|+1))$, und die Potentialabnahme ist von der Ordnung $O(n\log(n))$. Letzteres erhält man aus dem Lemma zu Satz 1 mit $w(i) \geq \frac{1}{k^2}$. Für die tatsächlichen Kosten der Sequenz ergibt sich die Summe

$$O\left(\sum_{j=1}^{m} \log(|i_j - f| + 1) + n \cdot \log(n)\right).$$

Die Abschätzung $O\left(m + n \cdot \log(n) + \sum_{j=1}^{m} \log(|i_j - f| + 1)\right)$ ist noch unbewiesen[17], *q.e.d.*

Satz 4: Wir numerieren die m Zugriffe von 1 bis m durch und definieren für jeden Zugriff j:
$t(j) :=\#$ der verschiedenen Schlüssel, auf die vor Zugriff j und seit dem letzten Zugriff auf Schlüssel i_j zugegriffen wurde. Dann ist $t(j) + 1$ die Position, die Schlüssel i_j in einer linearen Liste mit MTF–Regel einnehmen würde. Die tatsächliche Laufzeit für die m Zugriffe ist dann

$$O\left(n \cdot \log(n) + m + \sum_{j=1}^{m} \log(t(j) + 1)\right).$$

[17] 2Bdone

Bemerkung: Satz 1 impliziert, daß über eine genügend lange Folge von *access*–Operationen splay trees mindestens so effizient sind wie beliebige statische balancierte Suchbäume. Satz 2 läßt eine analoge Aussage für beliebige feste Suchbäume zu. Aus Satz 4 folgt, daß die Zugriffszeit auf einen Schlüssel abgeschätzt werden kann durch

log(1 + # versch. Schl., die seit letztem Zugriff auf <u>diesen</u> Schl. gesucht wurden).

Das heißt, die Schlüssel, auf die in letzter Zeit zugegriffen wurde, sind am leichtesten zu finden. Die Sätze 1 bis 4 lassen sich zu einem einzigen Satz vereinigen:

Satz 5: Die tatsächliche Laufzeit für m Zugriffe auf einen splay tree mit n Knoten ist

$$O\left(n \cdot \log(n) + m + \sum_{j=1}^{m} \log\left(\min\left(\left\{\frac{m}{q(i_j)}, |i_j - f| + 1, t(j) + 1\right\}\right)\right)\right).$$

Beweis: Definiere $w(i)$ als die Summe der Gewichte von i in den Beweisen zu Satz 2-4.

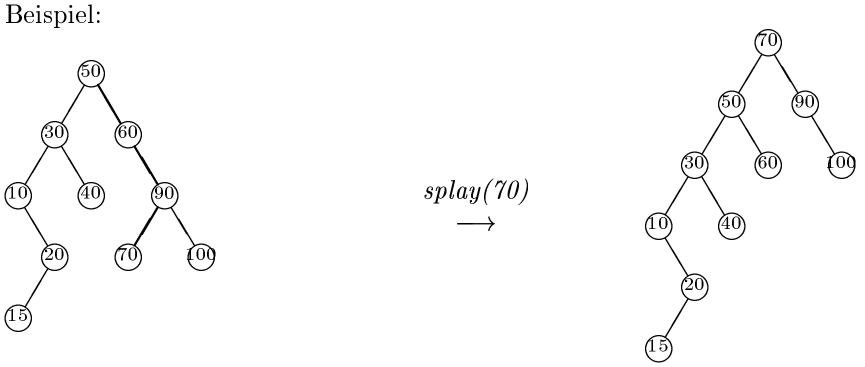
Bemerkung: Falls jeder Schlüssel mindestens einen Zugriff erfährt, so kann der Summand $n\log(n)$ fortgelassen werden.

3.8.5 Weitere Operationen auf splay trees

access(i,t): Falls Schlüssel i im Baum t enthalten ist, so wird ein Zeiger auf den i enthaltenden Knoten zurückgegeben. Falls der Schlüssel i nicht enthalten ist, so führt die Suche nach i zu einem Blatt, welches bzgl. der symmetrischen Ordnung entweder der unmittelbare Vorgänger oder der unmittelbare Nachfolger des gesuchten Knotens ist. Gemäß dieser Unterscheidung bezeichnen wir den Schlüssel j dieses Blattes y, welches den letzten Knoten des Suchpfads darstellt, als i^- oder i^+. Dann wird NIL zurückgegeben und *splay(y)* ausgeführt.

Implementierung: Durch binäre Suche wird der Knoten x mit $key(x) = i$ gesucht. Ist er gefunden, so wird *splay(x)* ausgeführt und ein Pointer auf die neue Wurzel x zurückgegeben. Ansonsten erhalten wir den unmittelbaren Vorgänger i^- oder Nachfolger i^+ am Blatt y, das den letzten Knoten auf dem Suchpfad darstellt, führen *splay(y)* aus und geben NIL zurück.

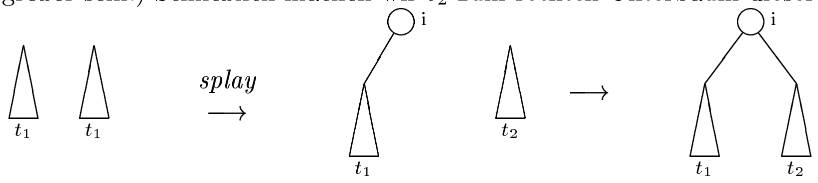
Beispiel:



Operation *access*(80,t)

join2(t_1,t_2): Die beiden Bäume t_1 und t_2 werden zu einem einzigen Baum vereinigt, vorausgesetzt, alle Schlüssel in t_1 sind kleiner als die Schlüssel in t_2.

Implementierung: Wir führen die *access*–Operation am größten Knoten i von t_1 durch. Dadurch wird der Knoten mit Schlüssel i zur Wurzel des Baums, und da die Sortiertheits–Invariante erhalten bleibt, besitzt er kein rechtes Kind. (Dies müßte größer sein.) Schließlich machen wir t_2 zum rechten Unterbaum dieser Wurzel.

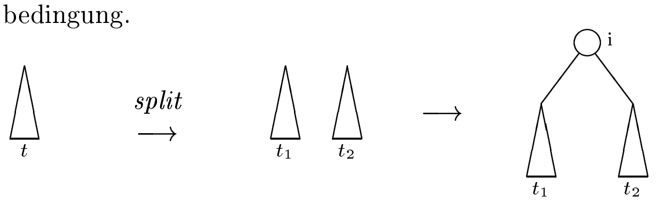


split(i,t)**:** Der Baum t wird in zwei splay trees t_1 und t_2 aufgespalten, wobei t_1 alle Schlüssel $\leq i$ enthält und t_2 alle Schlüssel $> i$.

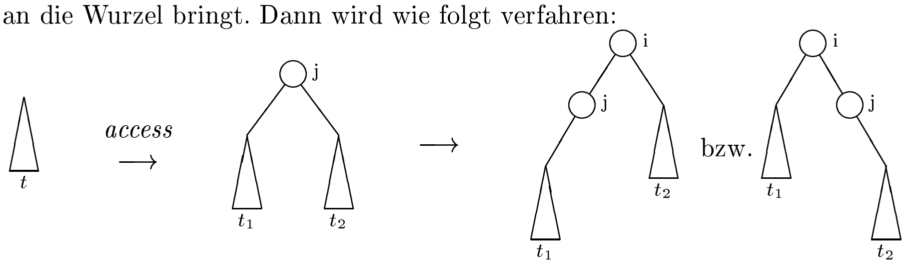
Implementierung: Wir führen *access(i,t)* aus, wodurch t zum neuen Baum t' mit Wurzel i' wird, und brechen die linke oder die rechte Kante an i' auf, abhängig davon, ob $i' \leq i$ oder $i' > i$.

insert(i,t)**:** Der Schlüssel i wird in den Baum eingefügt, vorausgesetzt, er ist noch nicht enthalten.

Implementierung: Zuerst wird *split(i,t)* ausgeführt und liefert die Bäume t_1 und t_2. Dann wird t durch einen neuen Baum ersetzt, dessen Wurzel i und dessen Unterbäume t_1 und t_2 sind. Diese Operation erhält offensichtlich die Sortiertheitsbedingung.

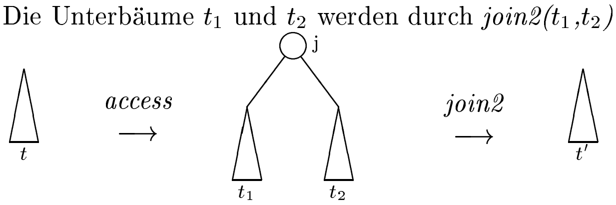


Eine alternative Implementierung führt zuerst *access(i,t)* aus, was $j = i^-$ bzw. i^+ an die Wurzel bringt. Dann wird wie folgt verfahren:



delete(i,t)**:** Der Schlüssel i im Baum t wird gelöscht, falls er enthalten ist.

Implementierung: Es wird *access(i,t)* ausgeführt, so daß i an die Wurzel gelangt. Die Unterbäume t_1 und t_2 werden durch *join2*(t_1,t_2) zu t verknüpft.



Alternativ kann auch der Schlüssel i im Baum t gesucht werden, was Knoten x liefert, und auf die Unterbäume t_1 und t_2 von x eine *join*–Operation angewendet werden. Sei y der Vater von x. Der Unterbaum mit Wurzel x wird durch das Ergebnis dieser *join*–Operation ersetzt. Schließlich wird *splay(y)* ausgeführt.

join3(t_1,i,t_2): Es wird ein Baum erzeugt, der die Elemente von t_1, i und die Elemente von t_2 enthält. Vorausgesetzt wird: $\forall\, z_1 \in t_1, z_2 \in t_2 : z_1 < i < z_2$.

Implementierung: Es wird einfach der ein Baum zurückgegeben, der i als Wurzel und t_1 und t_2 als linken bzw. rechten Unterbaum hat.

change-weight(i,t,Δ): Im Baum t soll die Veränderung $w(i) := w(i + \Delta)$ am Knoten x mit Schlüssel i vorgenommen werden. Dazu wird *splay(x)* ausgeführt und die Änderung vorgenommen.

Anmerkung: Alternativ kann auch die Operation *splay(i,t)* so erweitert werden, daß sie auch dann anwendbar ist, wenn i nicht im Baum t enthalten ist. Wie bei obigem *access* wird i gesucht. Bei erfolgloser Suche gelangt man zu einem Blatt y als letztem Knoten auf dem Suchpfad, dessen Schlüssel $j \in \{i^-, i^+\}$ ist. Es wird dann *splay(j)* ausgeführt. Dies hat den Vorteil, daß der Fall des Nichtenthaltenseins nicht bei der *access*–Operation berücksichtigt werden muß und *splay* universeller anwendbar wird.

3.8.6 Amortisierte Kosten der Operationen

Wir beginnen mit einer Menge von leeren splay trees und verlangen, daß grundsätzlich jeder Schlüssel in höchstens einem Baum enthalten sein kann. Das Potential *bal* einer Menge von Bäumen sei die Summe der Potentiale aller Bäume und der Logarithmen aller $w(i)$, für die i in keinem Baum enthalten ist.

Anmerkung: Die Potentialverkleinerung durch eine Folge von Operationen ist maximal

$$\sum_{i \in U} \log\left(\frac{w(i)}{w'(i)}\right),$$

wobei U das Universum aller Schlüssel ist und w bzw. w' die Gewichtsfunktionen vor bzw. nach der Sequenz von Operationen darstellen. Sind die Gewichte der Schlüssel unveränderlich, so ist diese Potentialveränderung stets nichtnegativ.

Satz: Die amortisierten Kosten der splay tree–Operationen haben die folgenden oberen Schranken, wobei W das Gesamtgewicht aller Schlüssel in den von der Operation betroffenen Bäumen ist:

access(i,t)	$\begin{cases} 3 \cdot \log\left(\frac{W}{w(i)}\right) + 1 & i \in t \\ 3 \cdot \log\left(\frac{W}{\min(\{w(i^-), w(i^+)\})}\right) + 1 & i \notin t \end{cases}$
join2(t_1,t_2)	$3 \cdot \log\left(\frac{W}{w(i)}\right) + O(1);\ i = \max(t_1)$
join3(t_1,i,t_2)	$3 \cdot \log\left(\frac{W}{w(i)}\right) + 1$
split(i,t)	$\begin{cases} 3 \cdot \log\left(\frac{W}{w(i)}\right) + O(1) & i \in t \\ 3 \cdot \log\left(\frac{W}{\min(\{w(i^-), w(i^+)\})}\right) + O(1) & i \notin t \end{cases}$
insert(i,t)	$3 \cdot \log\left(\frac{W - w(i)}{\min(\{w(i^-), w(i^+)\})}\right) + \log\left(\frac{W}{w(i)}\right) + O(1)$
delete(i,t)	$3 \cdot \log\left(\frac{W}{w(i)}\right) + 3 \cdot \log\left(\frac{W - w(i)}{w(i^-)}\right) + O(1)$
change-weight(i,t,Δ)	$3 \cdot \log\left(\frac{W + \Delta}{w(i)}\right) + O(1)$

Beweis: Satz 0 liefert die amortisierten Kosten fur *splay*. Da ansonsten keine Operationen verwendet werden, deren Kosten $O(1)$ übersteigen, bleibt nur die Potentialveränderung zu bestimmen. Sei s die Größenfunktion unmittelbar vor der Operation.

Dann beträgt der amortisierte Zeitaufwand für *access* und *split* nach Satz 0 höchstens $1+3\cdot\log\left(\frac{s(t)}{s(x)}\right)$, wobei am Knoten x die *splay*–Operation ausgeführt wird. Falls $i \in t$, so liegt i im Knoten x, und es gilt: $s(x) \geq w(i)$. Falls $i \notin t$, so liegt entweder i^- oder i^+ im Unterbaum mit Wurzel x, und es gilt: $s(x) \geq \min(\{w(i^-), w(i^+)\})$. Damit ist für *access* und *split* alles gezeigt. Die Schranke für *join2* folgt unmittelbar aus der für *access*. Die Zeit für *splay* ist höchstens $3\cdot\log\left(\frac{s(t_1)}{w(i)}\right)+1$, und die Potentialerhöhung ist $\log\left(\frac{s(t_1)+s(t_2)}{s(t_1)}\right) \leq 3\cdot\log\left(\frac{W}{s(t1)}\right)$ durch das Linken von t_1 und t_2, denn es ist $W \geq s(t_1)+s(t_2)$. Die Schranke für *insert* folgt ebenfalls aus der für *access*, denn die Potentialerhöhung durch das Hinzufügen der neuen Wurzel mit Schlüssel i beträgt $\log\left(\frac{s(t)+w(i)}{w(i)}\right) \leq \log\left(\frac{W}{w(i)}\right)$. Aus der Schranke für *access* und *join2* folgt sofort die Behauptung für *delete*, *q.e.d.*

Behauptung: Für die alternativen Implementierungen ergibt sich

- bei *insert(i,t)*: $3\cdot\log\left(\frac{W-w(i)}{\min(\{w(i^-),w(i^+)\})}\right) + \log\left(\frac{W}{w(i)}\right) + O(1)$
- bei *delete(i,t)*: $3\cdot\log\left(\frac{W-w(i)}{\min(\{w(i^-),w(i^+)\})}\right) + O(1)$.

Satz: Betrachte eine Folge von m *access*–Operationen, für die t die Anzahl der Vergleiche in einem optimalen und festen Suchbaum sei. Dann sind die Kosten für die *access*–Operationen bei Benutzung eines splay tree

$$O(t+n^2),$$

unabhängig von der Anfangskonfiguration des splay tree.

3.9 $O(\log(\log n))$– Warteschlangen

Priority queues, balancierte Bäume und ungeordnete (2,3)–Bäume sind Beispiele für Datenstrukturen, die eine Implementierung der Operationen *insert*, *min* und *deletemin* mit einem worst–case–Zeitaufwand von $O(\log(n))$ erlauben, wobei n die Anzahl der Elemente in der Datenstruktur ist. Eine Verbesserung ist möglich unter der Zusicherung, daß das Universum nur aus den ganzen Zahlen $0, 1, \cdots, (N-1)$ besteht. (Auf ähnliche Weise verkleinert sich unter dieser Voraussetzung die untere Schranke für das Sortieren von N Zahlen von $O(N\log(N))$ auf $O(N)$.) Die *van Emde Boas – Bäume (vEB–Bäume,* $O(\log(\log(N)))$*–Warteschlangen)* ermöglichen die Realisierung obiger Operationen mit einer Laufzeit von $O(\log(\log(N)))$ und sind damit den anderen Datenstrukturen überlegen, solange $n \geq \log(N)$ gilt.

3.9.1 k–Strukturen

Schreibweisen: Für $k \in \boldsymbol{N}$ mit $k \geq 2$ sei $k' := \left\lceil \frac{k}{2} \right\rceil$ und $k'' := \left\lfloor \frac{k}{2} \right\rfloor$.
Dann gilt: $k'+k'' = k$ bzw. $2^{k'}\cdot 2^{k''} = 2^k$. Für $x \in \boldsymbol{N_0}$ mit $x \in [0, 2^k-1]$ sei $x' := x$ div $2^{k''}$ und $x'' := x$ mod $2^{k''}$, so daß gilt: $x'\cdot 2^{k''} + x'' = x$.

Definition:
Eine *k–Struktur T zur Menge* $S = \{x_1, x_2, \cdots, x_m\} \subset [0, 2^k-1]$ ist (in PASCAL–Notation) ein `RECORD`, bestehend aus

(i) einer ganzen Zahl T.SIZE= $|S| = m$ sowie den Pointern T.MIN und T.MAX auf das minimale bzw. maximale Element in der Liste in (ii)

(ii) einer sortierten doppelt verketteten Liste T.LIST mit den Elementen $x_1, x_2, \cdots, x_m$ in aufsteigender Reihenfolge

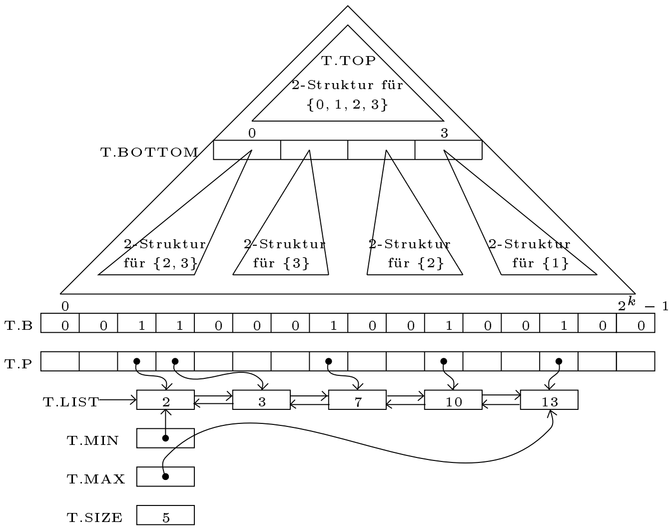
(iii) einem Bitvektor T.B$[0..2^k - 1]$ mit T.B$[x_i]$= 1 $\forall\ i \in \{1, \cdots, m\}$ und einem array T.P$[0..2^k - 1]$ von Pointern, wobei T.P$[x_i]$ auf das Element x_i in der Liste T.LIST zeigt

(iiii) einer k'–Struktur T.TOP und einem array T.BOTTOM$[0..2^{k'}-1]$ von k''–Strukturen. Diese haben jeweils denselben Aufbau wie eine k–Struktur, es liegt also ein rekursiver Aufbau vor. Dabei gilt:

- Für $m \leq 1$ sind die Strukturen T.TOP und T.BOTTOM leer. Das heißt, die SIZE–Komponenten sind 0, die Liste leer und die Bitvektoren nur mit 0en besetzt.
- Für $m \geq 2$ ist
  - T.TOP eine k'–Struktur für die Menge $\{x'_1, x'_2, \cdots, x'_m\}$, und
  - für jedes $x \in [0, 2^{k'} - 1]$ ist T.BOTTOM$[x]$ eine k''–Struktur für die Menge $\{x''_i | x'_i = x\}$.

Die hierzu dringend nötigen Erläuterungen folgen nach dem Beispiel, welches lediglich die Struktur des RECORD verdeutlichen soll:

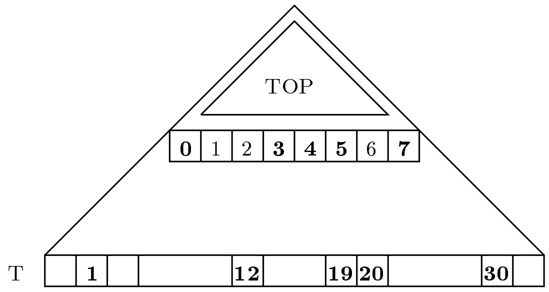
Beispiel: Sei $k = 4$ und $S := \{2, 3, 7, 10, 13\}$. Dann ist $k' = k'' = 2$, und die zu den Elementen x_i gehörigen x'_i sind 0, 0,1, 2 und 3. T.TOP ist eine 2–Struktur für die Menge $\{0, 1, 2, 3\}$, T.BOTTOM[0] ist eine 2–Struktur für $\{2, 3\}$, T.BOTTOM[1] ist eine 2–Struktur für $\{3\}$, T.BOTTOM[2] ist eine 2–Struktur für $\{2\}$ und T.BOTTOM[3] ist eine 2–Struktur für $\{1\}$.



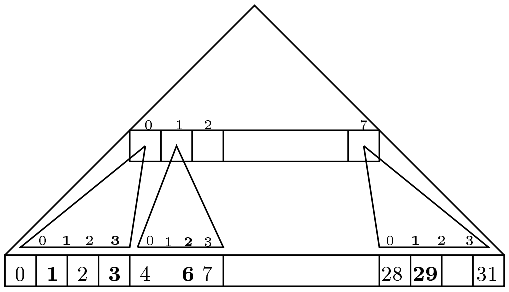
Erläuterungen zum rekursiven Aufbau:
Zuerst beachte man, daß der Wert von k nicht willkürlich zu wählen ist, sondern in

Abängigkeit von der gegebenen Menge S der Bedingung $S \subset [0, 2^k - 1]$ genügen muß. Der Aufbau einer k–Struktur wird sofort plausibel, wenn man bei gegebenem k, etwa $k = 5$, die maximale, also zur Menge $S = [0, 2^k - 1] = [0, 31]$ gehörige k–Struktur betrachtet. Allgemein interessieren wir uns bei einem 'Baum' der Höhe $k = \lceil \log \tilde{N} \rceil$ mit $\tilde{N}$ Blättern für den Vorfahren x' in Höhe $\lceil \frac{\log(\tilde{N})}{2} \rceil = k'$ eines Blattes x.
In der Höhe von x' gibt es $\leq 2^{\lceil \frac{\log(\tilde{N})}{2} \rceil} = 2^{k'}$ Knoten.
In unserem maximalen Fall sind das $2^3 = 8$ Stück. Der Übergang von x zu x' erfolgt mittels ganzzahliger Division durch $2^{k''}$. Da $S \subset \{0, \cdots, 2^k - 1\}$, gilt damit

$$x' \in \{0, \cdots, 2^{k'} - 1\} = \left\{ \left\lfloor \frac{0}{2^{k''}} \right\rfloor, \left\lfloor \frac{1}{2^{k''}} \right\rfloor, \cdots, \left\lfloor \frac{2^k - 1}{2^{k''}} \right\rfloor \right\}.$$



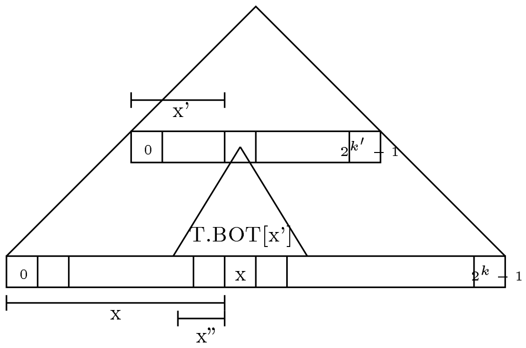
Weiterhin erhalten wir in jedem, nicht nur dem maximalen Fall *genau* $2^{k'}$ k''–Strukturen als T.BOTTOM. Das entspricht der maximalen Anzahl von Elementen in der T.TOP–Struktur. Man kann sich die Menge $\{0, 1, \cdots, 2^k - 1\}$ in genau diese Anzahl von gleichgroßen Abschnitten zerlegt denken; Jedes der Elemente von S ist dann einem dieser Abschnitte zugeordnet.



In jeder dieser $2^{k'}$ k''–Strukturen werden die Reste aus der Division der Elemente x durch $2^{k''}$ gespeichert, und zwar so, daß $x \bmod 2^{k''}$ in diejenige k''–Struktur abgelegt wird, die zu demjenigen Abschnitt von $\{0, 1, \cdots, 2^k - 1\}$ gehört, dem auch x angehört.
Anders gesagt, für jedes $x \in \{0, \cdots, 2^{k'-1}\}$ gilt: T.BOTTOM[x]= $\left\{ x_i'' \,\middle|\, \left\lfloor \frac{x_i}{2^{k''}} \right\rfloor = x \right\}$.
Damit liegen die Elemente der k''–Strukturen im Bereich $\{0, \cdots, 2^{k''} - 1\}$.
(Probe: In jeder dieser $2^{k'}$ Strukturen befinden sich im maximalen Fall $2^{k''}$ Elemente. Somit erhalten wir für diesen Fall wieder eine Gesamtzahl von $2^{k'} \cdot 2^{k''} = 2^k$ Elementen.)
Für die $\leq \tilde{N}$ Blätter erhalten wir maximal $O(\sqrt{\tilde{N}})$ Strukturen.

3.9.2 Anwendung der k–Strukturen: Die $succ$–Operation

Zweck der obigen Definitionen ist es uns zu ermöglichen, den Nachfolger *succ(x,T)* eines Elements x zu bestimmen. Folgende Figur sollte das Vorgehen verdeutlichen:



Wir entscheiden, ob x und *succ(x,T)* in Höhe $\left\lceil \frac{\log(\tilde{N})}{2} \right\rceil$ denselben Vorfahren haben. Dazu vergleichen wir x'' mit dem letzten Element in T.BOTTOM[x'].
Ist x'' <T.BOTTOM[x'].MAX↑, so ist die Antwort "ja", sonst "nein".

- Im "ja"–Fall haben wir *succ(x,T)* unter den $O(\sqrt{\tilde{N}})$ Elementen in T.BOTTOM[x'] zu suchen.
- Im "nein"–Fall sei z' :=*succ(x',T.TOP)*. Dann ist *succ(x,T)* das erste Element in der Liste zu T.BOTTOM[z']. Wieder erhalten wir eine Rückführung von $\tilde{N}$ auf $\sqrt{\tilde{N}}$, denn die Anzahl der x' ist $O(\sqrt{\tilde{N}})$.

Der hierdurch beschriebene Algorithmus hat eine Laufzeit, die gemäß der obigen Überlegungen der Rekursionsungleichung $T(n) \leq T(\sqrt{n}) + 1$ gehorcht. Expandieren der zugehörigen Rekursionsgleichung liefert die Abschätzung $T(N) = O(\log(\log(N)))$.

```
succ(x:[0..2^k − 1], T:k-Struktur);
if ((k=1) or (T.SIZE≤ 2)) then durchsuche T.LIST ;Laenge hoechstens 2
else if (x ≥T.MAX↑)
  then return ''EX. NICHT''
  else x':=x div 2^⌊k/2⌋; x'':=x mod 2^⌊k/2⌋;
    if ((T.TOP.B[x']=1) and (x'' <(T.BOTTOM[x']).MAX↑))
    then return x' · 2^⌊k/2⌋+succ(x'',T.BOTTOM[x'])
    else z':=succ(x',T.TOP);
      return z' · 2^⌊k/2⌋ + T.BOTTOM[z'].MIN↑
    fi
  fi
fi
```

Lemma: Obiges Programm berechnet *succ(x)* in Zeit $O(\lceil\log(k)\rceil)$.

Beweis:

Korrektheit: Der Nachfolger von x existiert genau dann nicht, wenn T leer ist, oder x das maximale Element in S ist. Sei dies nicht der Fall. Dann ergibt sich folgende Fallunterscheidung:

1. Fall: $\exists y \in S$ mit $y' = x'$; Dann befindet sich *succ(x'')* unter den z'' mit $z' = x'$. Existiert dieser Nachfolger, so sind wir fertig, denn für z'' =*succ(x'')* ist *succ(x)=z=$x' \cdot 2^{k''}$+succ(x'')=$z' \cdot 2^{k''} + z''$* (siehe Zeile 8 im Programm). Existiert nicht, so gehen wir vor wie in

2. Fall: $\nexists y \in S$ mit $y' = x'$; Dann ist z' =*succ(x')* in der TOP–Struktur zu suchen (Zeile 9), und das Ergebnis ist: $z' \cdot 2^{k''} + \min(\text{T.BOTTOM}[z'])$.

Zeitbedarf: Sei $R(k)$ die maximale Laufzeit für *succ* in einer k–Struktur. Dann ist $R(1)$ eine Konstante. Da rekursive Aufrufe stets für $\lceil \frac{k}{2} \rceil$– oder $\lfloor \frac{k}{2} \rfloor$–Strukturen erfolgen, und nur konstant viel Zeit außerhalb rekursiver Aufrufe verbraucht wird, gilt für R die Rekursionsgleichung

$$R(k) leq c + R\left(\left\lceil \frac{k}{2} \right\rceil\right).$$

Diese ergibt aufgelöst die Abschätzung $R(k) = c + c \cdot \lceil \log(k) \rceil$.

3.9.3 Weitere Operationen

insert:

```
insert(x:[0..2^k - 1], T:k-Struktur);
if (T.B[x]=0) then
  T.B[x]:=1;
  y:=succ(x,T); füge x neben y in T.LIST ein
  (Zielposition ohne Aufwand mit Hilfe von T.P[y] erhältlich)
  x':=x div 2^⌊k/2⌋; x'':=x mod 2^⌊k/2⌋;
  case T.SIZE of
  0: nop
  1: y:=anderes Element in T.LIST;
       insert(x',T.TOP);
       insert(y',T.TOP);
       insert(x'',T.BOTTOM[x']);
       insert(y'',T.BOTTOM[y']);
  ≥ 2: insert(x',T.TOP);
       insert(x'',T.BOTTOM[x']);
  esac;
  T.SIZE:=T.SIZE+1;
  T.MIN und T.MAX ggf. anpassen;
fi
```

Falls x in eine leere Struktur eingetragen werden soll, so müssen x' und x'' nicht in die TOP– bzw. BOTTOM–Struktur geschrieben werden; Wenn jedoch x das zweite Element wird, so ist dies auch für das erste Element nachzuholen.

Lemma: Das Programm `insert` fügt ein Element in Zeit $O(\lceil \log(k) \rceil) = O(\log(\log(N)))$ in die k–Struktur ein.

Beweis: Da die Korrektheit offensichtlich ist, zeigen wir nur die Schranke für die Laufzeit: Sei $R(k)$ die maximale Laufzeit für *insert* in einer k–Struktur. Dann ist $R(1)$ wieder eine Konstante c, und auch für das Einfügen in eine leere Struktur ist der Zeitaufwand $O(1)$. Die case–Anweisung führt auf folgende Fallunterscheidung:

Fall "0": Hierbei ist der Aufwand nur $O(1)$.

Fall "1": Die beiden rekursiven Aufrufe für x' und x'' fügen in eine leere Struktr ein. Daher ist ihr Zeitaufwand $O(1)$. Im Fall $x' = y'$ gilt dies auch für den Aufruf `insert(`y'`,T.TOP)`, denn y' ist bereits in `T.TOP` enthalten. Für den Fall $x' \neq y'$ wird y'' durch `insert(`y''`,T.BOTTOM[`y'`])` in eine leere Struktur eingefügt. Letztendlich kommt es also zu nur einem nichttrivialen rekursiven Aufruf.

Fall "≥ 2": Ist x' bereits in T.TOP enthalten, so erhält man für die erste Anweisung dieser Alternative die Komplexität $O(1)$, und ansonsten für den zweiten Aufruf von `insert`.

Für jeden dieser Fälle gilt damit die Abschätzung $S(k) \leq c + S(\lceil \frac{k}{2} \rceil)$ für den Zeitbedarf S der rekursiven Aufrufe. Darüberhinaus kostet die `succ`–Anweisung die Zeit $O(\lceil \log(k) \rceil)$. Für die gesamte Laufzeit ergibt sich die Rekursionsungleichung $R(k) \leq c + c\lceil \log(k) \rceil + R(\lceil \frac{k}{2} \rceil)$, was aufgelöst $O(\log^2(k))$ liefert. Das ist noch nicht die behauptete obere Schranke. Man erhält sie jedoch, wenn man berücksichtigt, daß die Folge der rekursiven Aufrufe in *succ* genau der Folge der nichttrivialen rekursiven Aufrufe bei *insert* entspricht. Ein Aufruf von `succ(`x`)` also liefert uns als rekursive Zwischenergebnisse alle Nachfolger von x, die wir benötigen. Wir speichern das Ergebnis des i-ten rekursiven Aufrufs von `succ` in $succ(x,T)^{(i)}$ ab, und beim i-ten Aufruf von `insert` zeigt uns dies den Nachfolger von $x^{(i)}$ in der Liste an. Der Gesamtaufwand zur Bestimmung aller Einstiegspunkte ('Nachfolger') in die Liste entspricht dem ersten Aufruf von *succ(x,T)*. Das liefert uns die Abschätzung

$$R(k) \leq 1 + R\left(\left\lceil \frac{k}{2} \right\rceil\right) = O(\log(k)) = O(\log\log(N)).$$

Delete:

```
delete(x:[0..2^k − 1], T:k--Struktur);
if T.B[x]=1
then T.B[x]:=0;
  lösche x in T.LIST;
  T.SIZE:=T.SIZE-1;
  case T.SIZE of
  0: nop;
  1: delete(x',T.TOP);
     delete(x'',T.BOTTOM[x']);
  ≥ 2: delete(x'', T.BOTTOM[x']);
     if T.BOTTOM[x'].SIZE=0
     then delete(x',T.TOP)
     fi
  esac
fi
```

Lemma: Dieses Programm löscht ein Element aus der k–Struktur in der Zeit

$$O(\log(k)) = O(\log(\log(N))).$$

Beweis: Der Beweis veräuft analog zum letzten. Wieder ergibt sich

$$R(k) = c + R\left(\left\lceil \frac{k}{2} \right\rceil\right) = O(\log(k))$$

.

Initialisierung:

Die Zeit zur Initialisierung einer k–Struktur ist proportional zu ihrem Platzbedarf $Sp(k)$. Aus der rekursiven Definition von k–Strukturen leiten wir sofort ab:

$Sp(1) = c$

$Sp(k) = c \cdot 2^k + Sp(\frac{k}{2}) + 2^{\frac{k}{2}} \cdot Sp(\frac{k}{2})$ für $k \geq 2$.

Behauptung: Es gilt: $Sp(k) \leq c \cdot 2^k \cdot \prod_{l=1}^{\log(k)} \left(1 + \frac{1}{2^{2^l}}\right) \cdot \log(k)$

Beweis: Induktion nach k; Es gilt:

$$\begin{aligned}
Sp(k) &\leq c \cdot 2^k + (1 + 2^{\frac{k}{2}}) \cdot c \cdot 2^{\frac{k}{2}} \cdot \prod_{l=1}^{\log(k)-1} \left(1 + \frac{1}{2^{2^l}}\right) \cdot \log\left(\frac{k}{2}\right) \\
&\leq c \cdot 2^k \cdot \prod_{l=1}^{\log(k)-1} \left(1 + \frac{1}{2^{2^l}}\right) \cdot \left(1 + \frac{1}{2^{\frac{k}{2}}}\right) \cdot \log(k) \\
&\leq c \cdot 2^k \cdot \prod_{l=1}^{\log(k)} \left(1 + \frac{1}{2^{2^l}}\right) \cdot \log(k)
\end{aligned}$$

Es folgt: $Sp(k) = O(2^k \cdot \log(k))$, denn $\prod_{l=1}^{\log(k)} \left(1 + \frac{1}{2^{2^l}}\right) = O(1)$.

Letzteres gilt wegen $\prod_{l=1}^{\log(k)} \left(1 + \frac{1}{2^{2^l}}\right) \leq 2^{\sum_{l=1}^{\log(k)} \log\left(1+\frac{1}{2^{2^l}}\right)} \leq 2^{\sum_{l=1}^{\infty} \frac{1}{2^{2^l}}} = O(1)$
und $\log(1 + x) \approx x$ für $x \to 0$.

Zusammenfassung: Die Operationen *insert*, *delete* und *delete-min* erfordern im vEB–Baum den Zeitaufwand $O(\log(\log(N)))$, *min* und *iselement* sind in Zeit $O(1)$ ausführbar. Das Initialisieren einer neuen k–Struktur kostet $O(N \cdot \log(\log(N)))$, und dieselbe Abschätzung gilt für den Platzbedarf einer Struktur.

3.9.4 Implementierung von vEB–Bäumen mit linearem Platz

Um den Platzbedarf von $O(N \cdot \log(\log(N)))$ zur Verwaltung einer Menge $S \subset [0..N-1]$ zu reduzieren, beachten wir, daß S im Universum $U = [0..N-1]$ veränderlich ist und teilen U in $\frac{N}{\log(\log(N))}$ Intervalle $U_i = [i \cdot \log(\log(N))..(i+1) \cdot \log\log(N) - 1]$ auf. Für alle i sei $S_i := S \cap U_i$, und s'_i codiere die Eigenschaft $(S_i \neq \{\})$. Wir verwalten die Menge aller Indizes i mit $s'_i = 1$ in einem vEB–Baum und benötigen zudem für jedes i eine Liste der Elemente in S_i. Die Summe der Längen dieser Listen ist höchstens N. Da der von uns verwendete van Emde Boas – Baum Zahlen bis zur Göße $\tilde{N} = \frac{N}{\log(\log(N))}$ verwaltet, und ein solcher Platz der Ordnung $O(\tilde{N} \cdot \log(\log(\tilde{N})))$ benötigt, erhalten wir hierfür eine Abschätzung von

$$O\left(\frac{N}{\log(\log(N))} \cdot \log\left(\log\left(\frac{N}{\log(\log(N))}\right)\right)\right).$$

Insgesamt ergibt sich:

$$\text{Platz} = \sum_{i=1}^{\frac{N}{\log(\log(N))}} O(\log(\log(N))) + O\left(\frac{N}{\log(\log(N))} \cdot \log\left(\log\left(\frac{N}{\log(\log(N))}\right)\right)\right)$$

$$= O(N) + O\left(\frac{N}{\log(\log(N))} \cdot \log\left(\log\left(\frac{N}{\log(\log(N))}\right)\right)\right)$$
$$= O(N).$$

Für die Laufzeit der Operationen bedenke man, daß bei Suchoperationen zusätzlich zum Auffinden des richtigen Index j die zu S_j gehörige Liste durchsucht werden muß. Da dies jedoch nur den Aufwand $O(\log(\log(N)))$ erfordert, ändert diese Implementierung nichts an der Komplexität der Operationen.

3.10 Union–Find–Strukturen

3.10.1 Grundlagen

Das "Union–Find–Problem" besteht darin, disjunkte Mengen unter den Operationen

- *makeset:* Schaffen einer neuen Menge, die nur das gegebene Element enthält
- *union:* Vereinigen zweier gegebener Mengen
- *find:* Bestimmung der Menge, die das gegebene Element enthält

zu verwalten. Zur Implementierung soll der Algorithmus in jeder der Mengen ein sogenanntes "*kanonisches Element*" unterhalten, das zur Identifikation der betreffenden Menge dient. Als ein solches Element kann ein beliebiges Datenelement verwendet werden, denn die Mengen sind als disjunkt vorausgesetzt, und somit ist ein Element für seine Menge stets charakteristisch.

Operationen:

> **makeset(x):** Schaffe eine neue Menge, die nur das Element x enthält. Dieses wird natürlich als kanonisches Element verwendet. Es ist in keiner anderen Menge enthalten.
>
> **find(x):** Gib das kanonische Element zu derjenigen Menge zurück, die x enthält.
>
> **union(x,y):** Vereinige die Mengen mit den kanonischen Elementen x und y und zerstöre die beiden Mengen. Wähle das kanonische Element für die Resultatmenge neu und gib es zurück. Voraussetzung: $x \neq y$.

Implementierung: Jede Menge wird als In–Tree dargestellt, dessen Knoten die Elemente enthalten. Die Wurzel enthält das kanonische Element. Somit besitzt jeder Knoten x einen Zeiger zu seinem Vater $p(x)$; Der Zeiger der Wurzel zeigt auf die Wurzel selbst. Unter Berücksichtigung dieser Konzepte kann nun die Implementierung der Funktionen angegeben werden:

> **makeset(x):** Definiere $p(x) := x$ und gib x zurück;
>
> **find(x):** Folge, ausgehend vom Knoten mit Inhalt x, den Zeigern bis zur Wurzel des Baums und gib das kanonische Element zurück;
>
> **union(x,y):** Definiere $p(x) := y$ und gib y zurück; Das heißt, der Baum mit Wurzel x wird an den mit Wurzel y angehängt, so daß y als ein weiteres Kind x erhält.

Bemerkung: Mit dieser Implementierung von *union* entartet ein Baum im worst case linear, so daß *find* die Komplexität $O(n)$ erhält, wenn n *makeset*–Operationen zum Aufbau des Baums verwendet wurden, der Baum also n Elemente enthält. Dies läßt sich durch Anwendung zweier Heuristiken, der *Pfadkompression* und dem Schema *Union–by–Rank* noch erheblich verbessern:

> **Pfadkompression:** Die Struktur des Baums wird während der *find(x)*–Operation verändert. Dabei wird ausgenutzt, daß der Algorithmus beim Hochlaufen der In–Pointer in Richtung Wurzel nicht nur für x, sondern auch für alle anderen Elemente auf dem Weg von x zur Wurzel erfährt, welcher Menge sie angehören. Deshalb werden ihre Zeiger nach Bestimmung der Wurzel direkt auf das kanonische Element gerichtet.
>
> **Union–by–Rank:** Hierfür wird eine Freiheit in der Spezifikation von *union* ausgenutzt, nämlich die Tatsache, daß es keine Rolle spielt, welcher der zu vereinigenden Bäume an welchen angehängt wird. Um die Komplexität von *find* zu reduzieren, wollen wir erreichen, daß die *union*–Operation die Bäume nicht unnötig hoch werden läßt. Dazu speichern wir zusammen mit jedem Knoten x eine ganze Zahl *Rang(x)*, die eine obere Schranke für die *Höhe* von x angibt. Es sei bereits jetzt darauf hingewiesen, daß die Aktualisierung dieser Werte außerordentlich wenig Aufwand bereitet: Nur solange x die Wurzel, also kanonisches Element ist, kann sich an seier Höhe überhaupt etwas ändern. (Ganz anders verhält es sich mit der Tiefe der einzelnen Knoten.)
> Bei *makeset(x)* wird *Rang(x)* mit 0 vorbelegt, bei *union(x,y)* werden *Rang(x)* und *Rang(y)* veglichen. Falls *Rang(x)*<*Rang(y)*, so wird $p(x) := y$ gesetzt und y zurückgegeben. Das bedeuet, daß grundsätzlich der weniger hohe Baum an den höheren angehängt wird. Falls aber *Rang(x)*=*Rang(y)*, so wird $p(x) := y$ gesetzt und danach *Rang(y)* um 1 erhöht. Die Erhöhung des Rangs ist bei den anderen Fällen nicht nötig.

Algorithmen:

makeset(x)**:**

```
p(x):=x;
Rang(x):=0;
```

find(x)**:** (ohne Pfadkompression)

```
if (x!=p(x)) then p(x):=find(p(x)) fi;
return p(x);
```

union(x,y)**:** (Union–by–Rank)

```
if (Rang(x)>Rang(y)) then
   p(y):=x; return x;
elsif (Rang(x)<Rang(y)) then
   p(x):=y; return y;
elsif (Rang(x)=Rang(y)) then
   p(x):=y; Rang(y)++; return y;
fi
```

3.10.2 Amortisierte Kostenanalyse für Pfadkompression

Definitionen und Beobachtungen

Wir betrachten eine Sequenz von m *union*– und *find*–Operationen, wobei wir als Ausgangssituation n einelementige Mengen annehmen wollen. Die Sequenz soll mit und ohne Pfadkompression ausgeführt werden, aber stets unter Verwendung der Heuristik Union–by–Rank.

Bemerkung (Grundlegende Beobachtungen):

(i) In beiden Fällen liegen zu jedem Zeitpunkt dieselben Mengen vor, repräsentiert durch einander entsprechende Bäume.

(ii) Diese Bäume haben dieselben Wurzeln.

(iii) Die bei Pfadkompression entstehenden Bäume sind weniger hoch und haben an einigen Knoten (die einmal Wurzel waren) höheren Verzweigungsgrad.

(iv) Bei *union* mit Pfadkompression wird Knoten x Vater von Knoten y genau dann, wenn dies auch ohne Pfadkompression geschieht. Nur bleibt im letzteren Fall x der Vater von y, während sich dies durch Pfadkompression ändern kann.

Folgende Schranken für die amortisierten Kosten gelten mit obigen Definitionen:

- $O(m \cdot \log(\log(n)))$; nachgewiesen von Fischer
- $O(m \cdot \log^*(n))$; gefunden von Hopcroft und Ullman[18]
- $\Theta((m+n) \cdot \alpha(n))$; tatsächliche Schranke laut Tarjan

Dabei ist α eine Inverse der Ackermannfunktion, die wie folgt definiert ist:

Definition (Ackermann–Funktion): Es gibt mehrere gebräuchliche aber nicht äquivalente Definitionen der Ackermann–Funktion, doch all diese Funktionen wachsen etwa gleichschnell. Wir geben hier eine an: Für $x, n \in \boldsymbol{N_0}$ ist

$$\begin{array}{rcl} A_0(x) & = & x+1 \\ A_{k+1}(x) & = & A_k^x(x), \end{array}$$

wobei $A_k^0 \equiv id$ und $A_k^i(x) = \underbrace{A_k(A_k(\cdots A_k}_{i\times}(x)\cdots))$, d.h. $A_k^i = \underbrace{A_k \circ A_k \circ \cdots \circ A_k}_{i\times}$.

Eigenschaften von $A_k(x)$: $A_k(x)$ ist für jeden Wert von k monoton wachsend in x, und stets gilt $A_k(x) \geq x$.
Eine der auffälligsten Eigenschaften von A_k ist ihr enormes Wachstum in k bei $x \geq 2$, während sie für $x \leq 1$ kleine Werte annimmt. A_k wächst im ersteren Fall sogar zu schnell, um noch LOOP–berechenbar zu sein. Es gilt dann

$$\begin{array}{rcll} A_0(x) & = & x+1 & \text{"erhöhe } x \text{ um 1"} \\ A_1(x) & = & 2x & \text{"erhöhe } x\text{-mal } x \text{ um 1"} \\ A_2(x) & = & 2^x \cdot x & \text{"multipliziere } x\text{-mal } x \text{ mit 2"} \\ A_3(x) & \geq & \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{x\times} & \end{array}$$

[18] Es gilt: $\log^{(0)}(n) = n$; $\log^{(i+1)}(n) = \log(\log^{(i)}(n))$ und $\log^*(n) = \min\left(\left\{i \mid \log^{(i)}(n) \leq 1\right\}\right)$.

So ist etwa für $x = 2$: $A_2(2) = 8$, $A_3(2) = 2^{11}$, $A_4(2) = \underbrace{2^{2^{2^{\cdot^{\cdot^{2}}}}}}_{2048\times}$.
Entsprechend langsam wächst jede Umkehrfunktion von $A_k(x)$ bei $x \geq 2$.

Definition: Wir erklären A als Funktion von nur einer Variablen k, nämlich

$$A(k) := A_k(2),$$

und bezeichnen sie als die *Ackermannfunktion*. Als ihre Inverse setzen wir

$$\alpha(n) := \min(\{k \in \boldsymbol{N_0} | A(2) \geq n\}).$$

Für alle praktischen Zwecke ist $\alpha(n)$ eine Konstante kleinergleich 4.

Anmerkung: Unser $O((m+n) \cdot \alpha(n))$ ist zwar nicht echt linear, kann jedoch für jede praktische Anwendung als linear angesehen werden, da α gewissermaßen constant ist.

Rang eines Knotens

Betrachte eine Sequenz der Länge m von *union*– und *find*–Operationen, angewandt auf die anfangs vorliegenden n einelementigen Mengen.

Definition: Es sei $T_t(x)$ der Unterbaum von x zum Zeitpunkt t. Dabei wird vereinbart, daß der *Zeitpunkt* t genau derjenige ist, zu dem die t–te Operation ausgeführt wird. Die Operationen werden *ohne* Pfadkompression ausgeführt.
Hiermit setzen wir

$$\text{Rang}(x) := 2 + \text{Höhe}(T_m(x)),$$

d.h. Rang(x) bezeichnet die um zwei erhöhte Länge eines längsten Pfades von x zu einem Blatt innerhalb des Baums, der nach Ausführung der *gesamten* Sequenz von Operationen vorliegt.

Bemerkungen:

(i) Solange x noch kanonisches Element (also ohne Vater) ist, kann sich Rang(x) noch erhöhen. Dies geschieht durch *union*–Operationen mit gleichhohen Bäumen. Sobald durch eine Vereinigung dem Knoten x ein Vater zugeordnet wird, bleibt sein Rang constant.

(ii) Falls keine Pfadkompression verwendet wird, ist die Höhe aller Bäume monoton wachsend bzgl. der Zeit t.

(iii) Die Beobachtungen (i) und (ii) zusammen ergeben folgende wichtige Aussage:
($\star$) Wird y zum Zeitpunkt t ein Nachfahr von x, so gilt zu jedem späteren Zeitpunkt $s > t$, daß Höhe$(T_s(x))$ > Höhe$(T_s(y))$, also insbesondere Höhe$(T_m(x))$ > Höhe$(T_m(y))$, und das heißt mit anderen Worten:

$$\text{Rang}(x) > \text{Rang}(y).$$

Lemma (Balancierung durch Union–by–Rank):
Bei Verwendung der Union–by–Rank Heuristik für die Vereinigungsoperation gilt stets:

$$|T_t(x)| \geq 2^{\text{Höhe}(T_t(x))},$$

d.h. Höhe$(T_t(x)) \leq \log(|T_t(x)|)$.

Beweis: Induktion nach t;

I) $T_0(x) = \{x\}$, und damit $|T_0(x)| = 1 \geq 2^0 = 2^{\text{Höhe}(T_0(x))}$.

II) Das Lemma gelte zum Zeitpunkt t.

1. Fall: Höhe$(T_{t+1}(x))$ = Höhe$(T_t(x))$. Dann gilt die Aussage erst recht, da

$$|T_{t+1}(x)| \geq |T_t(x)|.$$

2. Fall: Höhe$(T_{t+1}(x))$ = Höhe$(T_t(x)) + 1$ durch *union*. Dann wird $T_t(y)$ neuer Unterbaum von x, y wird ein neues Kind von x. Es ist $|T_t(y)| \geq 2^{\text{Höhe}(T_t(y))}$, und wegen Union–by–Rank gilt: $|T_t(y)| \leq |T_t(x)|$. Folglich:

$$\begin{aligned} |T_{t+1}(x)| &= |T_t(x)| + |T_t(y)| \\ &\geq 2^{\text{Höhe}(T_t(x))} + 2^{\text{Höhe}(T_t(y))} \\ &\geq 2^{\text{Höhe}(T_t(y))} + 2^{\text{Höhe}(T_t(y))} \\ &= 2^{\text{Höhe}(T_t(y))+1} = 2^{\text{Höhe}(T_{t+1}(y))}. \end{aligned}$$

Lemma:
Der maximale Rang eines Knotens nach Ausführung aller m Operationen der Sequenz ist $\leq \lfloor \log(n) \rfloor + 2$.

Beweis: Für jeden Knoten x gilt:

$$n \geq |T_m(x)| \geq 2^{\text{Höhe}(T_m(x))} \geq 2^{\text{Rang}(x)-2}.$$

Somit ist Rang$(x) \leq \lfloor \log(n) \rfloor + 2$.

Lemma: Es gilt für die Anzahl aller Knoten mit Rang r:

$$|\{x | \text{Rang}(x) = r\}| \leq \frac{n}{2^{r-2}}.$$

Beweis: Falls Rang(x) = Rang(y) für zwei Knoten x und y, so müssen $T_m(x)$ und $T_m(y)$ wegen $(\star)$ disjunkt sein, denn sonst wäre etwa x Vater von y und dann nach obiger Berechnung Rang$(x) >$ Rang(y). Folglich:

$$n \geq \left| \biguplus_{\text{Rang}(x)=r} T_m(x) \right| = \sum_{\text{Rang}(x)=r} |T_m(x)| \geq \sum_{\text{Rang}(x)=r} 2^{r-2} = 2^{r-2} \cdot |\{x | \text{Rang}(x) = r\}|.$$

Beweis: Nun werde die Sequenz *mit* Pfadkompression ausgeführt. Für einen Knoten x betrachten wir die 'Distanz' zu seinem Vorgänger $p(x)$ anhand der Rangdifferenz.

- Rang$(x) = 2 +$ Höhe$(T_m(x))$ ist per Definition zeitunabhängig, da t in der Bestimmungsgleichung nicht auftritt. Schließlich stützt sich der Rang auf den Zeitpunkt nach dem Ende aller Operationen.
- Rang$(p(x))$ dagegen ist *abhängig* von t. Die Funktion Rang ist zwar, wie eben festgestellt, zeitunabhängig, aber die Funktion p ist es nicht: Durch Pfadkompression kann der Vater $p(x) = p_t(x)$ eines Knotens x abhängig von der Zeit ändern.
- Rang$(p(x))$ kann sich bei wachsendem t wegen $(\star)$ nur erhöhen.

In der folgenden Diskussion betrachten wir die Bedingung

$$\text{Rang}(p(x)) \geq A_k(\text{Rang}(x))$$

Definition: Es sei $\delta(x) := \max(\{k \in \boldsymbol{N_0} | \text{Rang}(p(x)) \geq A_k(\text{Rang}(x))\})$.

Bemerkung: Der Wert von $\delta(x)$ ist abhängig von t und kann sich aufgrund der Pfadkompression mit wachsendem t erhöhen. Falls x einen Vorgänger besitzt, so ist die Existenz von $\delta(x)$ gesichert, denn schließlich ist $\text{Rang}(p(x)) \geq \text{Rang}(x) + 1 = A_0(\text{Rang}(X))$. Für $n \geq 5$ gilt

$$\delta(x) \leq \alpha(n) - 1.$$

Begründung: Für $\delta(x) = k$ gilt: $n > \lfloor \log(n) \rfloor + 2 \geq \text{Rang}(p(x)) \geq A_k(\text{Rang}(x)) \geq A_k(2)$, und folglich $k < \alpha(n)$.

Die Analyse

Da jede *union*–Operation die Zeit $O(1)$ kostet, ist der Aufwand für alle *union*–Operationen beschränkt durch $O(m)$.
Die Kosten für eine *find*–Operation sind proportional zur Länge des Pfades vom Ausgangsknoten zum Knoten mit dem kanonischen Element (der Wurzel). Dieser Pfad wird zweimal durchlaufen, das erste Mal zum Auffinden des kanonischen Elements und das zweite Mal, um für die Pfadkompression alle Pointer auf diesem Pfad zu verbiegen. Dies möge zusammen je eine Zeiteinheit pro Knoten des Pfades kosten. Die zum Knoten x gehörigen Kosten behandeln wir wie folgt:

- Hat x einen Vorfahren y (auf dem Pfad), so daß gilt: $\delta(y) = \delta(x)$, so wird die zu x gehörige Zeiteinheit dem Knoten x selbst zugerechnet.
- Andernfalls gehöre sie zur *find*–Operation. Somit werden jeder *find*–Operation maximal $\alpha(n)$ Zeiteinheiten berechnet, nämlich maximal eine für jeden der $\alpha(n)$ möglichen Werte von δ. Denn für jeden solchen Wert k von δ gilt: Nur die Zeiteinheit des letzten Knotens x auf dem Pfad mit der Eigenschaft $\delta(x) = k$ wird der *find*–Operation zugerechnet. Es liegen aber nur $\leq m$ *find*–Operationen vor, also ist die gesamte *find* zugerechnete Zeit $O(m \cdot \alpha(n))$.

Es interessiert ferner die Summe aller Zeiteinheiten, die einem Knoten x im Laufe der Rechnung berechnet werden. Zu einem Zeitpunkt t, zu dem x eine Zeiteinheit zugerechnet bekommt, hat x einen Vorfahren y mit $\delta(x) = \delta(y) =: k$, und es gilt:

$$\begin{aligned} \text{Rang}(p(x)) &\geq A_k(\text{Rang}(x)) \\ \text{Rang}(p(y)) &\geq A_k(\text{Rang}(y)). \end{aligned}$$

Es sei $i \geq 1$ derart, daß $\text{Rang}(p(x)) \geq A_k^i(\text{Rang}(x))$. Ein solches i existiert. Sei weiterhin y der letzte Knoten des Pfads. Dann gilt zur Zeit t, daß

$$\begin{aligned} \text{Rang}(y) &\geq \text{Rang}(p(y)) \\ &\geq A_k(\text{Rang}(y)) \\ &\geq A_k(\text{Rang}(p(x)) \\ &\geq A_k(A_k^i(\text{Rang}(x))) \\ &\geq A_k^{i+1}(\text{Rang}(x)). \end{aligned}$$

Zum Zeitpunkt $t + 1$ ist y der neue Vater von x, sodaß dann gilt:

$$\text{Rang}(p(x)) \geq A_k^{i+1}(\text{Rang}(x)).$$

Somit können dem Knoten x maximal Rang(x) Zeiteinheiten zugerechnet werden, bevor

$$\mathrm{Rang}(p(x)) \geq A_k^{\mathrm{Rang}(x)}(\mathrm{Rang}(X)) = A_{k+1}(\mathrm{Rang}(x))$$

und dann auch $\delta(x) \geq k+1$ gilt. Somit erhöht sich $\delta(x)$ spätestens dann um ≥ 1, wenn dem Knoten x Rang(x) Zeiteinheiten berechnet worden sind. Da sich $\delta(x)$ andererseits nur $(\alpha(n)-1)$-mal erhöht, können x insgesamt nur Rang$(x) \cdot \alpha(n)$ Zeiteinheiten angerechnet werden. Nach obigem Lemma sind das höchstens

$$r \cdot \alpha(n) \cdot \frac{n}{2^{r-2}} = n \cdot \alpha(n) \cdot \frac{r}{2^{r-2}}$$

Zeiteinheiten für alle Knoten mit Rang r. Summation über alle Werte von r ergibt folgende obere Schranke für die Gesamtkosten aller Knoten:

$$\sum_{r=0}^{\infty} n \cdot \alpha(n) \cdot \frac{r}{2^{r-2}} = n \cdot \alpha(n) \cdot \sum_{r=0}^{\infty} \frac{r}{2^{r-2}} = 8n\alpha(n).$$

Durch Summation obiger Abschätzung für die Anzahl der Zeiteinheiten, die den einzelnen Knoten bzw. der *find*–Operation zugerechnet werden, erhält man:

Satz: Eine Sequenz von m *union*– und *find*–Operationen, welche auf n einelementige Mengen angewendet werden, kosten $O((m+n)\cdot\alpha(n))$ Zeit.

3.11 Zugabe — Hashing–Arrays

3.11.1 Problemstellung

Gegeben sei eine Menge M von Elementen, von denen jedes durch einen Schlüssel k aus der Menge K bezeichnet sei. Die Problemstellung ist nicht neu: Wie muß die Speicherung der Elemente aus M organisiert werden, damit jedes anhand seines Schlüssels möglichst schnell lokalisiert werden kann? Gesucht ist also eine Abbildung $H : K \longrightarrow A$ von der Menge aller Schlüssel in den Adreßraum A der Maschine. Hierbei soll jedoch eine bisher nicht beachtete Schwierigkeit berücksichtigt werden: Die Menge K der möglichen Schüsselwerte ist wesentlich größer als der Adreßraum. Folglich kann die Abbildung H nicht injektiv sein, es gibt Schlüssel $k_1, k_2, \ldots$ mit $H(k_1) = H(k_2) = \cdots$. Wir werden sehen, daß aufgrund dieses Umstands die Speicherstelle eines Elements mit Schlüssel k von einem anderen Element mit einem anderen Schlüssel l besetzt sein kann. Die Hauptaufgabe in diesem Abschnitt wird sein, solche *Kollisionen* zu beheben.
Die Operationen, die wir unterstützen wollen, sind völlig kanonisch

- **iselement**
- **insert**
- **delete**

Vorgehen: Sei $n := |K|$ und $m := |A|$, und sei der Adreßraum o.B.d.A ein zusammenhängender Bereich mit den Adressen $0, 1, \ldots, (m-1)$. Dann verwenden wir ihn als *Hash–Tabelle* $T[0..m-1]$ und bezeichnen H als die *Hash–Funktion*. Um das Element mit Schlüsselwert k aufzufinden, führen wir folgende Schritte aus:

1. Berechnen der Adresse $H(k)$
2. Prüfen, ob der Schlüssel des Elements an dieser Adresse gleich k ist.

Das ist, wie oben angemerkt, nicht notwendig der Fall: Es kann eine Kollision vorliegen. Die Arten der Kollisionsbehandlung induzieren die Unterscheidung zwischen offenem und geschlossenem Hashing (siehe unten).

3.11.2 Wahl der Hash–Funktion

An die Hash–Funktion H werden von vornherein folgende Anforderungen gestellt:

(i) Gleichmäßige Verteilung der Schlüssel auf die Adressen $\{0, 1, \ldots, m-1\}$.

(ii) Efiziente Berechenbarkeit von H.

Der *Füllfaktor* der Hash–Tabelle ist definiert durch

$$\alpha := \frac{m}{n} = \frac{\#\text{ Adressen}}{\#\text{ Schlüssel}}$$

und gibt die durchschnittliche Anzahl der Schlüssel pro Adresse an.
Einige praktikable Hashing–Methoden sind:

1. **Teilermethode:** Wir wählen $H(k) := (k \bmod m) \in \{0, 1, \ldots, m-1\}$.
   Dabei soll die Grösse m des Adreßraums weder eine Zweierpotenz noch eine Zehnerpotenz oder von der Form $2^k - 1$ sein, sondern am besten eine Primzahl in der Nähe einer Zweierpotenz.

2. **Multiplikationsmethode:**
   Für ein $a \in]0, 1[$ sei $H(k) := \lfloor m \cdot \underbrace{(k \cdot a \bmod 1)}_{\text{Nachkomma–Anteil}} \rfloor$.
   Nach Knuth ist $\frac{\sqrt{5}-1}{2} \approx 0.618034$ ein günstiger Wert für a.

3. **Universelles Hashing:** Es wird zufällig eine Hash–Funktion aus einer endlichen Klasse von Funktionen ausgewählt. Dabei heißt eine Klasse $\mathcal{H}$ heißt *universell* genau dann wenn

   $$\forall k, l \in K \text{ mit } k \neq l \text{ gilt: } |\{h \in \mathcal{H} | h(k) = h(l)\}| = \frac{|\mathcal{H}|}{m}.$$

   Das bedeutet, daßin der Klasse $\mathcal{H}$ die Anzahl der Hash–Funktionen, die zwei bestimme Schlüssel kollidieren lassen, eindeutig festgelegt ist.

   Beobachtung: Sei $h \in \mathcal{H}$ zufällig. Bei einem fest vorgegebenen Schlüssel k ist die erwartete Anzahl von Kollisionen mit k höchstens $\frac{n-1}{m} > 1$.

   Satz: Sei die Schlüsselmenge $K := \{0, 1, \ldots, m-1\}^{r+1}$ und m eine Primzahl. Wähle

   $$\begin{aligned} \mathcal{H} \quad := \quad & \{H | H : \{0, \ldots, m-1\}^{r+1} \longrightarrow \{0, \ldots, m-1\}; \\ & H(k_0, \ldots, k_r) := \left(\sum_{i=0}^{r} (a_i \cdot k_i)\right) \bmod m; \\ & a_i \in \{0, \ldots, m-1\}\}, \end{aligned}$$

   dann ist $\mathcal{H}$ universell.

3.11.3 Methoden zur Kollisionsauflösung

(A) Geschlossenes Hashing:

Die Kollisionsbehebung erfolgt durch Verkettung: Statt die Elemente direkt in die Hash–Tabelle zu schreiben, speichert man an Adresse $H(k)$ einen Zeiger in eine Liste, in welche alle Elemente mit Schlüssel $k^{(i)}$ geschrieben werden, für die $H(k^{(1)}) = H(k^{(2)}) = \cdots$ Es liegt daher keine Kollisionsbehebung im eigentlichen Sinne vor, Kollisionen werden vielmehr völlig umgangen.

insert läßt sich stets in $O(1)$ implementieren — man fügt ein neues Element vorne in die geeignete Liste ein.

delete ist bei Verwendung von doppelt verketteten Listen in $O(1)$ realisierbar, wenn vorausgesetzt ist, daß ein Zeiger auf das zu löschende Element in der entsprechenden Liste gegeben ist.

iselement verlangt notwendig das Durchsuchen der gesamten zum Schlüssel k gehörigen Liste.

Satz: Seien die Wahrscheinlichkeiten dafür, daß ein bestimmtes Element mit Schlüssel k eine bestimmte Position in der Liste an Adresse $H(k)$ trifft, gleichverteilt, und sei wieder α der Füllgrad der Tabelle. Dann gilt bei erfolgreicher oder erfolgloser Suche (*iselement*):
Es werden $\Theta(1 + \alpha)$ Schritte benötigt.

Beweis: Bei erfolgloser Suche wird die gesamte Liste durchlaufen, bei erfolgreicher Suche nur durchschnittlich die halbe Liste. Die dadurch erzeugten Faktoren 1 bzw. $\frac{1}{2}$ sind in Θ berücksichtigt. Die durchschnitttliche Listenlänge ist der Quotient der Länge n aller Listen und der Anzahl m der Listen. Das ist gerade der Füllgrad α. Da für das Auswertwen von H noch der Aufwand $\Theta(1)$ anfällt, erhalten wir $\Theta(1 + \alpha)$.

(B) Offenes Hashing:

Bei dieser Methode kann es tatsächlich zu echten Kollisionen kommen, denn die Elemente werden nicht in Listen, sondern direkt in die Hash–Tabelle gespeichert. Wird bei *insert* oder *iselement*, angewendet auf Schlüssel k, ein Element mit Schlüssel $\neq k$ an Adresse $H(k)$ gefunden, so wird auf deterministische Weise eine alternative Adresse berechnet, an dieser Adresse nachgesehen, u.s.f. Dieser Vorgang heißt *Sondieren* (*probing*). Man verwendet Hash–Funktionen $H(k, i)$, die definiert sind durch

$$H(k, i) := \text{Adresse für Schlüssel } k \text{ nach } (i - 1) \text{ Kollisionen.}$$

Dabei unterscheidet man wieder mehrere verschieden effiziente Arten des Sondierens:

(a) **lineares Sondieren:** $H(k, i) := (H(k) - i) \bmod m$. Es wird für *iselement* solange rückwärts gesucht, bis entweder das Element mit Schlüssel k oder eine freie Position gefunden ist. Im letzteren Fall ist das gesuchte Element nicht in der Hash–Tabelle enthalten.

Problem: Es entstehen *primäre Häufungen* um diejenigen Schlüssel herum, die beim Einfügen keine Kollision hervorgerufen haben.

Satz: Die durchschnittliche Anzahl der Sondierungen ist
• bei erfolgreicher Suche: $\frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right)$ und
• bei erfolgloser Suche: $\frac{1}{2} \cdot \left(1 + \frac{1}{(1-\alpha)^2}\right)$.

α	erfolgreich	erfolglos
0.5	1.5	2.5
0.9	5.5	50.5
0.95	10.5	200.5

(b) **quadratisches Sondieren:** $H(k,i) := (H(k) + c_2 \cdot i^2 + c_1 \cdot i) \bmod m$

Problem: Es ist hierbei nicht sichergestellt, daß der gesamte Adreßraum besucht wird. Deshalb wird für das Einfügen eventuell kein freier Platz gefunden, obwohl die Tabelle noch nicht voll ist.

Satz: Die durchschnittliche Anzahl der Sondierungen ist
- bei erfolgreicher Suche: $\frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$ und
- bei erfolgloser Suche: $\frac{1}{1-\alpha} + \ln\left(\frac{1}{1-\alpha}\right)$.

α	erfolgreich	erfolglos
0.5	1.44	2.13
0.9	2.85	11.4
0.95	3.52	22.05

(c) **double hashing:** $H(k,i) := (H(k) + i \cdot \tilde{H}(k))$, wobei die Funktion $\tilde{H}$ relativ prim zu m gewählt werden muß.

α	erfolgreich	erfolglos
0.5	1.39	2
0.9	2.55	10
0.95	3.15	20

Referenzen:

Fredman M., Tarjan R., *"Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms"*, Journal of the ACM, Vol. 34, No. 3, Jul.1987

Kozen D., "The Design and Analysis of Algorithms", Springer–Verlag 1991

Mayr E., *Vorlesung "Effiziente Algorithmen" an der TU München im WS 1994/95*

Mehlhorn K., "Data Structures and Algorithms", Springer–Verlag 1984

Sleator D., Tarjan R., *"Amortized Efficiency of List Update Rules"*, Communications of the ACM Vol. 28, No. 2, Feb.1985

Sleator D., Tarjan R., *"Self–Adjusting Binary Search Trees"*, Journal of the ACM, Vol.32, No.3, Jul.1985

Tarjan R., *"Efficiency of a Goot but Not Linear Set Union Algorithm"*, Journal of the ACM, Vol. 22, No. 2, Apr.1975

Tarjan R., *"Data Structures and Network Algorithms"*, SIAM CBMS-NSF 44

van Emde Boas P., Kaas R., Zijlstra E., *Design and Implementation of an Efficient Priority Queue*, Mathematical Systems Theory 10, 1977